Applying Functional Programming Theory to the Design of Workflow Engines

Peter M. Kelly

January 2011

A dissertation submitted to the School of Computer Science of
The University of Adelaide for the degree of Doctor of Philosophy

Supervisors:

Dr. Paul D. Coddington

Dr. Andrew L. Wendelborn

Contents

Abstract

Workflow languages are a form of high-level programming language designed for coordinating tasks implemented by different pieces of software, often executed across multiple computers using technologies such as web services. Advantages of workflow languages include automatic parallelisation, built-in support for accessing services, and simple programming models that abstract away many of the complexities associated with distributed and parallel programming. In this thesis, we focus on data-oriented workflow languages, in which all computation is free of side effects.

Despite their advantages, existing workflow languages sacrifice support for internal computation and data manipulation, in an attempt to provide programming models that are simple to understand and contain implicit parallelism. These limitations inconvenience users, who are forced to define additional components in separate scripting languages whenever they need to implement programming logic that cannot be expressed in the workflow language itself.

In this thesis, we propose the use of functional programming as a model for data-oriented workflow languages. Functional programming languages are both highly expressive and amenable to automatic parallelisation. Our approach combines the coordination facilities of workflow languages with the computation facilities of functional programming languages, allowing both aspects of a workflow to be expressed in the one language.

We have designed and implemented a small functional language called ELC, which extends lambda calculus with a minimal set of features necessary for practical implementation of workflows. ELC can either be used directly, or as a compilation target for other workflow languages. Using this approach, we developed a compiler for XQuery, extended with support for web services. XQuery's native support for XML processing makes it well-suited for manipulating the XML data produced and consumed by web services. Both languages make it easy to develop complex workflows involving arbitrary computation and data manipulation.

Our workflow engine, NReduce, uses parallel graph reduction to execute workflows. It supports both *orchestration*, where a central node coordinates all service invocation, and *choreography*, where coordination, scheduling, and data transfer are carried out in a decentralised manner across multiple nodes. The details of orchestration and choreography are abstracted away from the programmer by the workflow engine. In both cases, parallel invocation of services is managed in a completely automatic manner, without any explicit direction from the programmer.

Our study includes an in-depth analysis of performance issues of relevance to our approach. This includes a discussion of performance problems encountered during our implementation work, and an explanation of the solutions we have devised to these. Such issues are likely to be of relevance to others developing workflow engines based on a similar model. We also benchmark our system using a range of workflows, demonstrating high levels of performance and scalability.

Declaration

This work contains no material which has been accepted for the award of any other degree or diploma in any university or other tertiary institution to Peter Kelly and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text.

I give consent to this copy of my thesis, when deposited in the University Library, being made available for loan and photocopying, subject to the provisions of the Copyright Act 1968.

I also give permission for the digital version of my thesis to be made available on the web, via the University's digital research repository, the Library catalogue, the Australasian Digital Theses Program (ADTP) and also through web search engines, unless permission has been granted by the University to restrict access for a period of time.

Peter Kelly

Acknowledgments

Firstly, I would like to express my sincere gratitude to my supervisors, Paul and Andrew, for their invaluable guidance and assistance with my work. Both have been excellent supervisors, and provided a great deal of insightful advice and feedback that has guided my project. Their unique combination of skills — in particular, Andrew's expertise in programming language theory and parallel functional programming, and Paul's background in high performance computing and scientific applications — has played a crucial role in the development of the key ideas behind my project. I am indebted to them for the many ways in which they have contributed to my development as a researcher.

Secondly, to those who have accompanied me on this journey in various ways — Alison, Asilah, Brad, Daniel, Donglai, Jane, Joanna, Joseph, Miranda, Nick, Nicole, Peter, Wei, Yan, and Yuval — thanks for making this whole process a more enjoyable one. I value the many opportunities we've had to exchange thoughts, ideas, and experiences. You have all been a great help to me in many ways, both personally and professionally.

Thirdly, many thanks go to my parents, who have done so much over the years to support and encourage me in my studies, and teach me the value of working hard to pursue my goals. To fully describe the ways in which they have helped me get to where I am today would take another three hundred pages. It is to them that I dedicate this thesis.

Finally, I am very grateful to the School of Computer Science for giving me the opportunity to pursue a PhD, and for providing me with financial support and an excellent working environment in which to carry out my studies. I would also like to acknowledge eResearch SA for providing me with access to the high performance computing facilities necessary to run my experiments.

Related publications

1. Peter M. Kelly, Paul D. Coddington, and Andrew L. Wendelborn. Distributed, parallel web service orchestration using XSLT. In *1st IEEE International Conference on e-Science and Grid Computing*, Melbourne, Australia, December 2005.

2. Peter M. Kelly, Paul D. Coddington, and Andrew L. Wendelborn. A simplified approach to web service development. In *4th Australasian Symposium on Grid Computing and e-Research (AusGrid 2006)*, Hobart, Australia, January 2006.

3. Peter M. Kelly, Paul D. Coddington, and Andrew L. Wendelborn. Compilation of XSLT into dataflow graphs for web service composition. In *6th IEEE International Symposium on Cluster Computing and the Grid (CCGrid06)*, Singapore, May 2006.

4. Peter M. Kelly, Paul D. Coddington, and Andrew L. Wendelborn. A distributed virtual machine for parallel graph reduction. In *8th International Conference on Parallel and Distributed Computing Applications and Technologies (PDCAT '07)*, Adelaide, Australia, December 2007.

5. Peter M. Kelly, Paul D. Coddington, and Andrew L. Wendelborn. Lambda calculus as a workflow model. *Concurrency and Computation: Practice and Experience*, 21(16):1999–2017, July 2009.

All of the software developed in this project is available from the following URL, under the terms of the GNU Lesser General Public License:

`http://nreduce.sourceforge.net/`

For Mum & Dad

Chapter 1

Introduction

Distributed computing [9] is an approach to solving large-scale computational problems by splitting applications into lots of individual tasks, which run in parallel across many different computers. It is common in many scenarios involving large amounts of data and computation, such as analysing results from scientific experiments, or deriving intelligence from information stored in corporate data warehouses. This thesis focuses on a particular class of distributed computing technology, *workflow languages* [329], which provide high-level programming models for composing tasks.

An important advantage of workflow languages is that they provide abstractions which hide complex implementation details such as socket programming and thread synchronisation, enabling programmers to focus instead on the high-level functionality of their application. Typically, parallelism in a workflow is detected automatically based on data dependencies, and tasks are scheduled to machines by the runtime system, without any involvement from the programmer.

Workflows automate procedures that users would otherwise have to carry out manually [85]. For example, a scientist may take the results of an experiment, run several independent analysis procedures on these results, and then combine the analysis results using another procedure which produces a report. Often, the programs used for such processing are exposed as *web services* [305]. By constructing a workflow that specifies which services are to be used and how data should be passed between them, these processes could be automated, and the analysis procedures could all run in parallel, reducing the amount of time required to produce results.

In this thesis, our interest is in making it easier to develop complex, large-scale workflows. We approach this by examining the requirements of workflows, reviewing existing work to see how well it meets these requirements, and then considering design approaches which build on existing programming language theory to support workflow development. In particular, we seek to answer the following question: Is it possible to reconcile the *coordination* philosophy of workflow languages with the *computation* facilities of general-purpose programming languages? Combining both aspects into a single language would simplify development of workflows by allowing the programmer to achieve their goals without having to repeatedly switch between two different languages.

1.1 Case study: Image similarity search

As an example of a situation in which a data processing workflow would be useful, consider a web site that hosts a large collection of images, and provides a keyword-based search function which returns a list of images matching a user's query. Each image in the search results contains a link which takes the user to a page displaying other images that are visually similar, in terms of properties such as shape, texture, and colour. For performance reasons, it is impractical to compute the set of related images every time a user clicks on such a link. The site therefore relies on a pre-computed index which records the relationships between different images, enabling the information to be retrieved quickly. For each image, the index stores the list of related images, ordered by their similarity score.

Construction of the index is computationally expensive, and must be carried out using a large collection of machines working in parallel. In order to support this and other batch processing required for the site, the machines are all configured with a service that provides general-purpose image processing operations which can be used for a wide variety of purposes. The indexing process is implemented as a workflow which uses these services to analyse and compare images.

The steps involved with the indexing process are as follows:

- Analyse all available images, producing a set of image profiles summarising their most important visual properties
- For each image:
  - Compare its profile with that of every other image, producing a list of (image name, similarity score) tuples
  - Filter the resulting list to remove all but the closest matches
  - Sort the list so that the closest matches come first
- Produce an index file which lists the name of each image, the names of related images, and the similarity scores

Two parts of this process are carried out by service operations: image analysis, and profile comparison. The rest of the logic exists in a workflow, which is developed separately from the services. The workflow coordination is carried out by a single machine, which invokes the analysis and comparison operations on other machines using remote procedure calls.

1.2 Language requirements

The example discussed above is representative of a large class of workflows used for data processing, many of which are used to support scientific research. These workflows require programming environments which provide the following features:

- **Remote service access.** For workflows such as this one which utilise a service oriented architecture, task execution is achieved by communicating with other machines to request that particular operations be performed. This requires support for network communication and data encoding within the language implementation. Preferably, the programmer should be able to focus on the input and output data exchanged with the services, without having to manually deal with system calls required to transmit the data and manage multiple concurrent network connections.

- **Parallelism.** In order to harness all of the available computing resources, it must be possible to invoke multiple service operations in parallel. In the example workflow, all of the analysis operations can execute in parallel with each other, as can all of the the comparison operations. Ideally, the programmer should not have to manually deal with threads and synchronisation primitives, given the difficult and error-prone nature of these language constructs.

- **Data structures.** The values sent to and received from tasks may be structured objects, and possibly arranged in other data structures such as lists and trees. It is therefore necessary to support these structures in the workflow language. The example workflow uses lists of objects representing image profiles and comparison results. Each object within these lists consists of multiple values, such as the name and profile data of an image.

- **Internal computation.** In addition to the computation carried out by external tasks, it is sometimes necessary to include small amounts of computational logic within the workflow itself. This requires built-in operations such as arithmetic, string manipulation, and boolean logic. In the example workflow, the sorting and filtering processes require the programmer to specify comparison and predicate functions which determine the ordering and inclusion of items in the result list by comparing numeric values. Generation of the index file requires the ability to construct a stream of text or binary data out of a collection of individual values.

- **Control flow.** The structure of a complex workflow often involves control flow logic such as iteration, conditional branching, user-defined functions, and recursion. In the example workflow, nested loops are needed for comparing pairs of images and producing the output. If the sorting and filtering logic is implemented in the same language as the workflow, then both of these rely on iteration and conditional branching.

1.3 Programming models for workflows

When writing a workflow, the programmer must make a decision about which language to use. One option is to use a general-purpose programming language such as C or Java, which are well-suited to expressing the computational aspects of the workflow. However, parallel programming in these languages is difficult and error-prone, and management of multiple concurrent network connections requires complex, low-level socket programming. Another option is to use distributed computing middleware such as a workflow engine to coordinate the execution of tasks. This would make the parallelism aspect easy, but may not permit all aspects of the workflow to be expressed. In the example workflow, sorting, filtering, and data serialisation may need to be implemented separately in external tasks, complicating the development process.

What is really needed is a solution that combines the features of general-purpose programming languages with those of distributed computing systems, making parallel and distributed programming easy to achieve without sacrificing support for language expressiveness. Importantly, this requires a programming model in which it is possible to automatically detect the parallelism present in the workflow, so that the programmer does not have to concern themselves with issues such as thread creation and synchronisation.

In this thesis, we focus exclusively on *data-oriented* workflows, which use execution models such as dataflow [160] and graph reduction [161]. In a data-oriented workflow, all computation is expressed as a transformation from input to output. Individual operations and the workflow as a whole are free of side effects, greatly simplifying the exploitation of parallelism. In contrast, *state-oriented* workflows involve operations with side effects, such as modifying entries in a database. These workflows require programming models in which the programmer explicitly defines the sequence in which operations must be performed, to ensure they are carried out in the correct order. Some literature uses the terms *scientific workflow* for the former and *business workflow* for the latter, in reference to the scenarios in which each is normally used. Section 2.3.1 discusses the differences between the two classes in more detail.

In addition to the aspects of the programming and execution models mentioned above, there are other considerations that are important for workflow systems. One concern is how errors and timeouts are dealt with — distributed computing inherently includes the possibility of failures, due to network disconnections or machine crashes, and clients should deal with these in a graceful manner. Another concern is execution management — the tracking, monitoring, and managing of computations. Although these issues are important, they are outside the scope of this thesis, and are left for future work. Sections 2.3.5 and 2.3.6 discuss these issues in a little more detail, and we suggest approaches for addressing fault tolerance in Section 8.4.3.

1.4 Functional programming

Functional programming languages [144] operate on a model in which all computation is free of side effects. This means that the only outcome of evaluating an expression is the result it produces; existing data cannot be modified. This style of computation can be described as purely data-oriented, because the focus is on producing output, rather than modifying state. This is a property shared with many data-oriented workflow languages.

One of the most important benefits of the functional programming model is that it is theoretically possible to automatically determine which parts of the program can run in parallel. The ability to do this fundamentally relies on the lack of side effects. If an expression A + B is evaluated, then A and B can both be safely run in parallel, because it is guaranteed that neither will be able to modify any state that is accessed by the other. If side effects were permitted, then the expression B might rely on a global variable assignment or data structure modification performed by A, and thus its result would depend on the order in which the two expressions were evaluated.

Parallelism is detected by constructing a graph from an expression, and using data dependencies to determine when tasks are independent of each other and can thus be run in parallel. This is conceptually equivalent to the way in which parallelism is detected in data-oriented workflow languages. For example, the expression E = A + B has one dependency between between E and A, and another dependency between E and B. Since there are no dependencies between A and B, their relative order of execution is unconstrained, and it is therefore possible to evaluate them in parallel.

The theoretical model underlying functional languages is *lambda calculus* [67], an abstract, Turing-complete [307] model of computation. It is theoretically possible to express any computable function in terms of lambda calculus, and functional languages are essentially extensions of the model providing built-in functions and data types, as well as various forms of syntactic sugar. Using lambda calculus as the basis for a language permits programmers to express arbitrary computation in the language, and opens up the possibility of automatically extracting parallelism from the code.

1.5 Thesis overview

In this thesis, we describe an approach to designing and implementing workflow languages which incorporates ideas from functional programming to provide a programming model in which it is possible to express both computation and coordination. The ability to include internal computation within a workflow is important, since services often don't provide all of the functionality needed to solve a given problem. By including language constructs for expressing internal computation, and executing this logic efficiently, a workflow language can make it easier to develop complex workflows.

Our basic programming model, ELC, supports arbitrary computation and data manipulation, by providing a small set of primitive operations and data types. It exposes services at the level of byte streams, enabling programmers to build up their own abstractions for supporting different service protocols and data formats. For web services, we support the use of XQuery for defining workflows. XQuery is well-suited to interacting with web services, as it provides native language support for XML, making it very straightforward to manipulate data values exchanged with web services. Also, since XQuery is a side effect free functional language, it is amenable to automatic parallelisation, and can be translated into ELC.

In comparison to other workflow languages, our approach removes the need for programmers to switch between different languages when implementing the coordination and computation aspects of a workflow. Our solution provides automatic parallelisation, built-in primitives for service access, support for arbitrary computation, and a concise syntax. We arrived at this solution after examining the relationships between functional and workflow languages, and realising that they have a great deal in common, even though there has previously been little attention paid to combining ideas from the two areas.

Specifically, the three main aspects of our work are the following:

1. **A workflow model based on lambda calculus (Chapter 3).** We define a programming language called ELC, which stands for *extended lambda calculus*. This builds on the basic lambda calculus model, adding standard features common to most functional languages, as well as features specifically tailored towards workflows. In particular, it includes a purely data-oriented mechanism for invoking remote services based on modeling services as functions which transform from an input stream to an output stream. ELC meets all of the requirements described in Section 1.2; it makes it easy to invoke multiple services in parallel, while also permitting programmers to express arbitrary computation within the workflow itself. The language is very similar to Scheme [184].

2. **A distributed virtual machine for executing workflows (Chapter 4).** To demonstrate the practicality of this model, we designed and implemented NReduce, a virtual machine based on parallel graph reduction, for executing workflows written in ELC. It features automatic detection of parallelism, efficient execution of fine-grained computation, a highly optimised user-mode threading mechanism, asynchronous I/O operations, a purely data-oriented abstraction of TCP connections, efficient representation of lists and data streams, and dynamic throttling of requests. The virtual machine also includes support for a distributed execution mode, whereby different nodes of the virtual machine are co-located with services on different machines, enabling data to be transferred directly between the services instead of via a central coordinating node, saving bandwidth.

3. **An XQuery implementation supporting web services (Chapter 5).** Many workflows involve tasks exposed as web services, which use XML for encoding all invocation messages and data values. To make data manipulation easy for such workflows, we developed an implementation of XQuery [43], an existing side effect free functional language which provides native support for XML processing. Although the language specification itself does not include support for web services, we implemented an extension which enables XQuery programs to invoke web services, thereby enabling it to be used as a workflow language. Our XQuery implementation is built on top of ELC and NReduce, thus enabling it to take advantage of their benefits, such as automatic parallelisation. This aspect of our project illustrates how our programming model and virtual machine can be used as a platform for supporting other high-level workflow languages.

1.6 Thesis outline

Chapter 2 contains a literature survey covering distributed computing and functional programming. We examine the capabilities and limitations of existing workflow languages and functional languages, as well as the relationships between the two.

Chapter 3 presents our programming model for expressing workflows. This model adapts existing theory from the world of functional programming to support the specific needs of workflows within the context of a service oriented architecture

Chapter 4 discusses a virtual machine which implements our programming model. It is designed to efficiently execute workflows involving large numbers of tasks and a medium amount of internal computation expressed directly within a workflow. The virtual machine is based on *parallel graph reduction* [24], an existing technique for implementing parallel functional languages.

Chapter 5 illustrates how the virtual machine and programming model from the previous two chapters can be used to support other workflow languages. We implement XQuery [43], a functional language designed for XML processing, which we extend with the ability to define workflows involving web services.

Chapter 6 discusses a range of problems we encountered during our implementation related to the management of parallelism, and solutions we developed to these problems to achieve efficient execution. Issues addressed include the impact of strict evaluation as compared to lazy evaluation, the throttling of service requests, and load balancing techniques.

Chapter 7 reports on a performance evaluation of our virtual machine and XQuery implementation in a range of different scenarios. We present results showing that our work successfully achieves the goals of large-scale parallelism and efficient execution of internal computation.

Chapter 8 concludes the thesis, providing a summary of the contributions made and a discussion of future work.

Chapter 2

Background and Related Work

In this thesis, we are concerned with the use of distributed computing [9, 197, 91] for harnessing many different computers together in parallel to achieve improved performance for computationally intensive applications. This use of distributed computing is of particular importance in *e-Science* [139], which is the use of software infrastructure to coordinate scientific experiments and analysis of results. Many systems have been developed over the years with the goal of making it easier to develop programs which distribute their computation among many machines and take advantage of parallelism. This thesis contributes to research on programming models for distributed computing by applying ideas from functional programming to the design and implementation of workflow languages. This chapter reviews a selection of existing work related to distributed computing and functional programming and justifies the motivation for our work.

In this thesis, we focus on a particular style of distributed computing in which the programmer divides their application into a collection of *tasks*, each of which is an atomic unit of work that takes some input data, performs some computation on it, and produces a result. Each task may run on a separate computer, in parallel with others, thus allowing the work to be completed faster than if only a single machine were to be used. In the systems we consider here, tasks do not communicate with each other during execution, though in some cases their results may be made available as input data to subsequently executed tasks. The distributed computing system is responsible for maintaining information about the set of tasks that comprise an application, and scheduling them in a suitable manner, taking dependencies into account where appropriate. Ideally, all details of scheduling, distribution, and fault tolerance are abstracted away from the programmer.

In Sections 2.1 – 2.3, we consider three categories of distributed computing systems. Those based on *independent tasks* are suitable for embarrassingly parallel applications consisting of a large collection of tasks which can each run completely independently. Systems based on a *fixed task structure* permit limited forms of dependencies between tasks, and coordinate execution and data transfer between the tasks according to this structure. *Workflow languages* [85] provide more flexible ways of arranging tasks by allowing the programmer to define arbitrary data dependencies between tasks, and use simple control flow constructs such as conditionals and loops.

A workflow language is essentially a form of high-level programming language in which each primitive operation corresponds to a task. Workflow languages are used for coordination — arranging multiple tasks together to achieve some goal, distribution — accessing services which are only available at specific sites, and parallelism — harnessing the power of multiple computers in parallel for performance reasons. These aspects, and other aims of workflow languages, are discussed in Section 2.3.2.

Existing workflow languages are lacking in expressiveness, support for fine-grained computation, and a solid grounding in existing programming language theory. Although a small number of business workflow language models are based on formal models[1], workflow languages often lack the flexibility and features present in most other types of programming languages. We address these concerns by adopting ideas from *functional programming languages* [144], which are well-suited to expressing data-oriented workflows due to their lack of side effects, which greatly simplifies the exploitation of parallelism.

Section 2.4 reviews concepts and existing work on functional programming, and examines the promising but little-explored synergy between the two areas. Existing functional languages however do not possess all of the features necessary for workflow execution, notably automatic parallelisation and support for I/O within pure expressions, which is necessary for accessing services. Chapters 3 and 4 discuss our adaptation of the functional programming model to the design and implementation of a workflow engine.

Web services [305] are of significant relevance to workflows, as they are a useful mechanism for exposing tasks on remote machines and providing invocation mechanisms that a workflow engine can use to request task execution via the Internet. In Section 2.5 we examine the basic concepts behind web services and some of the existing standards. We also discuss asynchronous remote procedure calls, an important technique used by workflow engines to invoke multiple services in parallel.

In considering the need for more powerful workflow languages, we look to *XQuery* [43] as a candidate. Although XQuery does not presently support accessing web services, existing work has proposed extensions to the language which add this capability. Our interest in XQuery as a workflow language is due to its native support for XML processing, ability to express many forms of computation, and potential for automatic parallelisation. Section 2.6 discusses existing approaches to integrating web service support into XQuery, some of which have guided our own work. The implementation of XQuery we have developed is discussed in Chapter 5.

This chapter aims to provide the reader with a sufficient understanding of both the requirements and limitations of existing workflow systems, as well as the opportunities offered by functional programming as a simple yet highly expressive model for data-oriented workflows. Later chapters of this thesis discuss our vision for workflow languages which leverages existing, well-established theory and implementation techniques from the functional programming community.

[1]One example is YAWL [311], which is based on Petri nets [234]

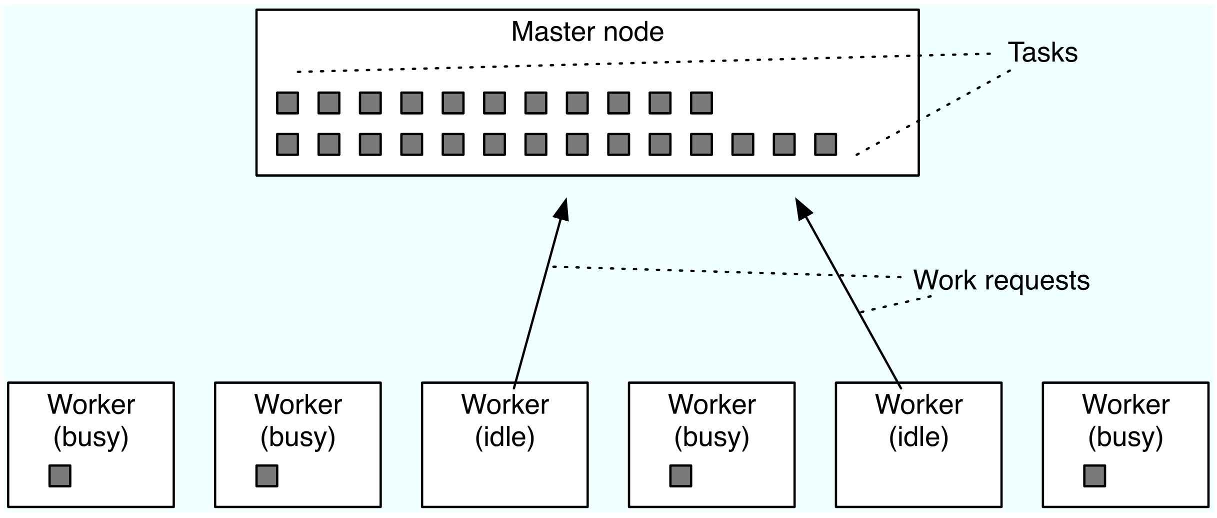


Figure 2.1: Task farming

2.1 Independent tasks

When each of the tasks in an application is able to be executed independently, distributing them among machines is conceptually very simple, and is achieved using a technique called *task farming*. A *master node* maintains a database recording information about each task, and tracks which ones are pending, in progress, or completed. The master node schedules tasks out to *worker nodes*, each of which provides a runtime environment capable of executing the tasks. This architecture is depicted in Figure 2.1.

If the worker nodes are dedicated, the master node may perform scheduling decisions itself, based its knowledge of each node's status. If the worker nodes are non-dedicated, i.e. may potentially be running other programs, then they may explicitly notify the master whenever they are idle and are ready to receive tasks for execution.

Three common styles of task arrangement are supported by task farming systems. The first is the general case of a batch queuing system, where each task can run a different program, and use different inputs. The tasks may all be related to solving one particular problem, or may be entirely unrelated, and submitted by different users. Batch queuing systems like PBS [138], Maui [154], and Sun Grid Engine [113] support this mode of operation. The second style of arrangement is *data parallelism*, where all tasks run the same program, but are supplied with different inputs. The inputs to these tasks may be of arbitrary size and format. The third style is *parameter sweep*, a special case of data parallelism where the inputs are numeric parameters which range between a specified minimum and maximum value, which is particularly common for tasks involving simulations. All three styles of task arrangement can be realised by a master node scheduling the tasks to worker nodes.

Fault tolerance is straightforward to implement for task farming systems. If execution fails due to a worker node crashing or becoming disconnected from the network, the tasks it was running can be rescheduled to another machine. Since this can be managed by the task queuing mechanism, a user who submits a collection of tasks to be executed does not need to manually deal with any failures; they simply need to wait until such time as all tasks have been completed, with re-executions occurring as necessary.

The task farming paradigm can be implemented on top of numerous parallel computing architectures. With shared memory systems [195], each processor acts as a worker node, and fetches tasks from a shared data structure. With clusters [57], the head node manages the task queue and keeps track of which machines are busy or idle, scheduling tasks to each as appropriate. With networks of non-dedicated workstations [92], each machine sends out an explicit work request to the master whenever it is neither executing a job nor being used interactively by a user. With volunteer computing [262], where the nodes may be located anywhere in the world, a similar model is used in which the task queue is managed by a server on the Internet which hands out jobs to volunteer machines.

One of the first systems for distributed computing was Condor [296], which can be used for either dedicated clusters or networks of user workstations. Each machine that is willing to execute tasks acts as a *resource*, and registers itself with a *matchmaker* service, which provides the ability to find machines to execute tasks based on the requirements of the tasks and the capabilities of machines. Tasks are submitted to an *agent*, which implements a job queue, and schedules tasks to resources based on those that match the requirements of the job. The combination of agents, matchmakers, and resources together constitutes a distributed system capable of managing the execution of large numbers of tasks across different machines. Condor includes a workflow tool called DAGMan [76], which we discuss in Section 2.3.7.1.

A well-known task farming system for Internet computing is SETI@home [15], which is designed for scanning signals obtained from radio telescopes for signs of extra-terrestrial life. This project was initiated by researchers seeking a low-cost alternative to traditional supercomputing, who decided to utilise volunteer machines across the Internet. By setting up a central server to manage the task queue, and encouraging volunteers to donate their spare CPU cycles, they were able to build a network of millions of worker nodes to execute their tasks.

SETI@home proved successful enough that the infrastructure was extended to cater for the computational needs of other scientific experiments. The resulting system, known as BOINC (Berkeley Open Infrastructure for Network Computing) [14], allows scientists to set up their own volunteer computing projects in which their application code is installed on worker machines provided by participants on the Internet. Each task corresponds to a piece of input data that is supplied to an instance of the application, with the results reported back to the central server. BOINC enables the owners of worker machines to control which projects they participate in, and how much priority each project has in terms of the CPU cycles that it may use. This infrastructure has been successfully used for scientific projects in a range of areas, including astronomy, physics, earth sciences, and medicine.

Sun Grid Engine [113] is designed for distributing jobs among machines in either a single cluster, or multiple clusters located in different parts of an organisation. It is designed for large-scale enterprises, which often have complex administration requirements, such as the need to define custom policies for different types of jobs and users. Job queues and scheduling are handled by a single master node, to which users may submit jobs. The master interacts with a set of execution nodes, which are responsible for executing jobs. A sophisticated scheduling mechanism takes into account various aspects of the execution nodes and jobs, such that jobs with specific hardware requirements can be matched to nodes that meet those requirements. Other properties of jobs that can affect scheduling include priority, waiting time, and the identity of the submitter. Sun Grid Engine supports *batch jobs*, which run on a single node, *parallel jobs*, which run across multiple execution nodes, and *interactive jobs*, in which the user may interact with the program while it is running.

XGrid [18] is a distributed computing system developed by Apple, and supplied as part of recent versions of OS X. It is based on a similar architecture to Condor, whereby desktop machines can make themselves available for executing tasks that are assigned to them from a queue. A master node is configured to accept job requests from authorised users, and distribute them to worker nodes. As it is primarily aimed at networks of workstations, XGrid by default only executes jobs when the machine is not currently being utilised by a regular user; for example, machines could be used for running jobs at night time, and normal desktop applications during the day.

Alchemi [213] is a workstation-based distributed computing system based on the .NET framework. Unlike most other similar systems, tasks are implemented as a .NET class, instead of an executable program. Each task must implement a specific interface that is used by the worker nodes to invoke a computation. As with XGrid, a master node is used to manage the queue of tasks waiting to be executed, and to allow users to submit new tasks. Because of the security mechanisms built-in to the .NET framework, Alchemi is able to protect worker nodes from malicious tasks by preventing privileged operations from being executed by untrusted code.

Nimrod/G [5] is a task farming system designed specifically for parameter sweep applications. The collection of tasks to be executed is derived from an input specification provided by a user which specifies the parameters along with ranges and step sizes for which the application code is to be evaluated. Nimrod/G includes a graphical monitoring tool which allows users to see the status of the collection of tasks, which is useful for monitoring progress. The system was originally designed for single clusters, but has been extended to cover wide-area configurations in which machines may be located at different sites.

Entropia [64] is a commercial system targeted at Windows-based desktop machines. Like other such systems, it is based on a centralised job queue, and worker nodes that volunteer their CPU time. A job submitted to the queue consists of a single application and multiple inputs. Each task is an invocation of this application with one of the inputs, and different tasks may be carried out by separate worker machines. The main innovation in Entropia is that it uses sandboxing techniques to allow unmodified Win32 binaries to be safely executed on worker nodes, such that they cannot perform privileged operations, such as accessing arbitrary parts of the file system. This is a particularly important feature in environments where untrusted code may be submitted to the system.

A somewhat related piece of software to the task farming systems described above is the Globus toolkit [106]. It is designed as a generic mechanism for launching tasks on machines, and provides a consistent interface on top of various job queuing systems. In this way, a client that is able to communicate with Globus can submit a task description in a specific format, and this will be translated into whatever syntax or interactions is necessary to launch a job on a particular batch queuing system. Globus can thus be used as a standardised layer on top of heterogeneous job queues on machines located at different sites, though it is primarily aimed at cluster-based queues rather than networks of workstations. Recent versions of the middleware have been extended with other high-level functionality, such as web services and data management, which are useful in conjunction with task execution facilities.

The task farming systems described above are useful for applications that can be expressed as a collection of independent units of work. However, some applications require tighter coupling between tasks, such that the outputs of some tasks can be used as inputs to others. This requires the distributed computing middleware to provide facilities for expressing these relationships, and taking them into account during execution, with regards to execution ordering and data transfer. In the next section, we discuss middleware that uses a fixed set of relationships between tasks, and in Section 2.3, we look at *workflow systems*, which provide the programmer with a greater degree of flexibility with respect to task relationships.

2.2 Fixed task structure

In some applications, data dependencies exist between tasks. For applications that follow certain common design patterns, there are systems available which are able to manage the parallelism and distribution of work and data transfer in a way that is optimised for these specific patterns. Although these systems are less flexible than workflow languages, which permit arbitrary task arrangements to be defined, they benefit from the fact that the knowledge about the application's structure may be taken into account during the design of the system, often enabling optimisations that would not otherwise be possible.

Perhaps the best known example of this category is MapReduce [83], a middleware system created by Google to manage many of their internal data processing needs, such as web indexing. MapReduce uses two types of tasks; *map* and *reduce*, inspired by their counterparts in functional programming. The map tasks consist of a large number of invocations of a `map` function, which accepts a key/value pair, and returns a list of key/value pairs, the latter of which may be of a different type to the input. Reduce tasks involve invocations of a `reduce` function, which accepts a key and a list of values corresponding to those output by the `map` function, and returns a list of values. The type signatures of these two functions are as follows:

$$\begin{array}{lll} \text{map} & (k_1, v_1) & \rightarrow \ \text{list}\,(k_2, v_2) \\ \text{reduce} & (k_2, \text{list}\,(v_2)) & \rightarrow \ \text{list}\,(v_2) \end{array}$$

The programmer supplies implementations of these two functions, and the middleware takes care of executing them across a collection of computers for a set of input data. The data itself must be in one of several specific formats, from which it is possible to extract a set of key/value pairs, though it is possible to write custom parsers for formats that are not already supported by the system. The map tasks and reduce tasks typically run on separate sets of computers; large-scale parallelism is possible in each case, due to the fact that many different keys are typically processed.

MapReduce has proven useful for a wide range of compute-intensive applications. In many cases, it is possible to choose the `map` and `reduce` functions, as well as the keys and values, in such a way that a solution to a given problem can be expressed within this structure. Several other implementations of the model have since been developed, including Hadoop [78], Phoenix [255], and MapReduce on Cell [82] — the latter two targeting shared memory systems.

OptimalGrid [176] is another distributed computing system based on fixed task relationships, but this time aimed specifically at applications based on the *finite element method* [276]. In this model, the tasks are arranged in a multi-dimensional matrix, where each task communicates with its neighbours. Rather than running separately, all tasks conceptually run at the same time, and communicate intermediate data with their neighbours during execution. An iterative process occurs in which each task performs some local computation, and then exchanges values with its neighbours. This is particularly useful for simulations in which some physical structure is modeled as a set of individual objects or areas which are connected to each other, and whose behaviour has an effect on adjacent objects.

Since the tasks are typically fine-grained, OptimalGrid does not use a separate thread for each, but executes each task sequentially during each iteration. Because it has knowledge of the relationships between the tasks, it can place neighbours on the same machine wherever possible, only requiring network communication in the case where a task's neighbours reside on separate machines. The placement of tasks on worker nodes is based on a load balancing algorithm that considers both computation and communication costs, and dynamically monitors performance during runtime to re-assign tasks where necessary.

2.3 Workflows

In contrast to the systems described in the previous section, *workflow systems* [329, 318, 30, 85] enable a programmer to arrange the tasks to be executed in a variety of different ways. This is important for applications in which data dependencies exist between the tasks, and a flexible mechanism of arranging them is necessary so that data produced by some tasks can be consumed by others. Workflow languages are a form of high-level programming language in which operations correspond to tasks that are executed by external pieces of software, usually remotely. Design patterns commonly used in workflows have been surveyed by van der Aalst et. al. [312], Bharathi et. al. [37], Pautasso and Alonso [248], and Yildiz, Guabtni, and Ngu [327].

The concept of computer-driven workflows originally comes from business process automation systems, which focus on the management of work within an organisation involving both automated tasks and human activities [141]. In more recent times, the idea has been applied to entirely computer-based processing, and more sophisticated models have been developed. In particular, data processing workflows have become very common in e-Science [295]. The development of workflow systems for e-Science was originally motivated by the need to automatically coordinate processes that would otherwise have to be controlled manually, such as running simulation and analysis tools, copying files, and accessing interactive web forms [325]. We discuss the differences between the types of workflow models used for business process automation and those used for scientific data processing in Section 2.3.1.

Workflows in e-Science are typically based on a data-oriented model of computation in which the dependencies between tasks are made explicit, and may be used to automatically detect opportunities for parallelism. This automatic parallelisation is a major advantage of workflow languages over imperative programming languages [230]. The ability for the programmer to define a high-level specification of their program without having to concern themselves with details like message passing and thread synchronisation is highly attractive.

Most scientific workflow languages provide built-in support for accessing web services and other RPC (Remote Procedure Call)-style services. This enables tasks implemented using different programming languages or platforms to be used within a workflow in a platform-independent manner. It also enables the workflow to incorporate functionality from publicly accessible services available via the Internet. Parallel execution is possible when accessing remote services located on multiple machines, and provides a form of latency hiding. Other types of tasks are often supported as well, such as external command-line programs, calls to Java functions, code snippets written in scripting languages, and submission of jobs to batch queuing systems.

A weakness in the design of many scientific workflow languages is that they often lack a solid grounding in programming language theory. It is rare to find workflow literature which references work done in the 1970s and 1980s on important topics such as compiler design [8, 161], remote procedure calls [292, 278], functional programming [124, 231, 144], and other aspects of language theory [124]. Much of the work that has been done in these areas over the years has been ignored by the workflow community, resulting in much reinvention of ideas, particularly in terms of control flow constructs, list processing, and language syntax.

We argue that when designing any new language, it is vital to understand what has come before, and that workflow languages must be considered within the broader context of distributed and parallel programming languages. Specifically, we shall discuss in Section 2.4 the ideas of *functional programming*, *lambda calculus*, and *graph reduction*, which are ideally suited towards workflows, but have seen almost no attention from the workflow research community.

2.3.1 Types of workflows

In discussing workflow technology, it is important to make the distinction between *state-oriented* and *data-oriented* workflows. Most literature refers to these as *business* and *scientific* workflows [210, 211, 28], but we prefer the former categorisation as it relates to a distinguishing characteristic of the respective programming models, rather than the context in which these types of workflows are normally used.

In this project, we are interested solely in data-oriented workflows; these are most commonly used for coordinating experiments and data analysis and integration in scientific research [327], but do have applications in industry as well. State-oriented workflows must be expressed in a workflow language based on an *imperative* programming model, as they require the ability to manipulate global state and enforce a specific order of execution. Data-oriented workflows lack side effects, and are therefore more appropriately expressed in workflow languages based on the principles of *dataflow* or *functional programming.*

State-oriented workflows involve the manipulation of state through an explicitly ordered sequence of instructions. Multiple threads of control may optionally be used, often via pairs of fork/join constructs, but each thread contains a specific sequence of instructions, and may need synchronisation with other threads. Invocation of a service often involves state changes, such as modifications to a database. Because of these side effects, state-oriented workflows are often executed in a transactional manner, whereby failure of a service is handled by either rolling back the transaction or carrying out a series of compensation actions to reverse the changes [93, 131]. State-oriented workflows are primarily used for business process automation, which involves interaction with multiple systems or business partners to complete a business transaction [81]. The most well-known workflow language of this type is BPEL (Business Process Execution Language) [16], which is an industry standard supported by many different vendors.

Data-oriented workflows contain a set of tasks, connected together via data dependencies. Each task takes a set of input values, and produces an output value. Importantly, all operations and the workflow as a whole are free of side effects. This means that the only outcome of executing a task is the result data it produces; no state changes occur at all. Fault tolerance is much simpler to implement in such a model, since tasks can be safely re-executed if a failure occurs, and there is no need for transactional semantics. Parallelism is also implicit, because it is possible for the workflow engine to inspect the workflow graph and work out which tasks can be executed in parallel based purely on data dependencies; the lack of side effects guarantees that no two concurrently-executing tasks can interfere with each other. There is no standard data-oriented workflow language, but several well-known ones exist, which we discuss in Section 2.3.7. BPEL can theoretically be used for data-oriented workflows, but is poorly suited for this purpose due to its explicit, imperative style of parallelism, which is why languages specifically designed for data-oriented workflows are more appropriate.

2.3.2 Aims of scientific workflow languages

Workflow languages are distinguished from general-purpose programming languages by their focus on being easy to use for those with little programming expertise, and the fact that they are intended solely for expressing *coordination* rather than *computation* — with the latter implemented by external services. Workflow engines allow computation to be distributed across multiple sites, either for the purpose of utilising services that are only available at specific locations, or to harness the performance benefits that come from running multiple tasks in parallel. Let us consider each of these aspects in turn:

- **Ease of use.** Workflows are normally edited using a graphical user interface which displays the workflow structure as a graph. These interfaces are targeted at domain experts such as scientists, who know what they want to achieve but lack the programming skills necessary to use a general-purpose programming language. Although a graphical representation can simplify things somewhat, the user still has to understand the semantics of the language, and possess the problem-solving skills necessary to derive an appropriate workflow structure for the task at hand. Thus, workflow development can still be considered a form of programming, even if the user may not think of it that way.

- **Coordination.** Workflow languages are designed solely for *coordination* rather than *computation*. The primitive operations of a workflow language correspond to complex tasks implemented in general-purpose programming languages. Examples of tasks include simulations, data analysis, and report generation. Coordination refers to the arrangement of these tasks so that they are executed in an appropriate order to satisfy data dependencies, and to specify which input data is supplied to which tasks. Workflow languages are used for "programming in the large" [87], whereas general-purpose programming languages are used for "programming in the small".

- **Distribution.** Workflows often make use of services provided by computers located at different sites. In some cases, this is necessary because some services are only available at particular sites, due to being proprietary or involving controlled access to private databases. In other cases, the services are replicated across multiple sites, and distribution is used to divide the workload among different computers so they can execute tasks in parallel, reducing the time required to complete the workflow.

- **Parallelism.** This aspect is important for those workflows that involve large amounts of computation, split among different tasks. When distribution is used for improving performance through parallelism, the workflow system is used as a means of coordinating the execution of tasks on remote machines in a manner that relieves the programmer of the responsibility of managing this execution manually. The high-level nature of workflow languages makes expression of parallelism much easier and less error-prone in comparison to the parallel programming constructs found in general-purpose programming languages.

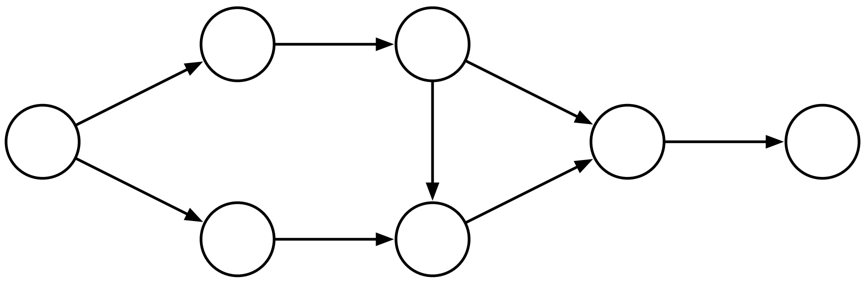


Figure 2.2: Task dependency graph

Given the opportunities for parallelism afforded by the side effect free nature of data-oriented workflow languages, a major emphasis of our work is in exploiting this parallelism for performance reasons. Much of the engineering effort invested into the development of our virtual machine, described in Chapters 4 and 6, has been focused on the ability to automatically manage large numbers of parallel tasks without imposing unacceptable performance overheads for locally-executed code. Additionally, Chapter 7 presents a detailed evaluation of how our system scales performance-wise with respect to the number of machines. However, our work does not preclude the use of distribution simply for the purpose of using services which are only located at specific sites.

2.3.3 Programming models

Existing data-oriented workflow languages are based on the *dataflow* model of computation [160], in which a workflow is represented as a *task dependency graph*. This is a directed graph in which nodes correspond to tasks, and edges indicate data dependencies. The graph specifies a partial order of execution, by constraining each task to begin only after those it depends on have completed. It is assumed that each task is free of side effects, and that its output depends only on its inputs. Tasks with no direct or indirect dependencies between them may thus be safely executed in parallel. This enables opportunities for parallel execution to be determined automatically by analysing the dependencies within the graph. Some workflow systems also permit *control flow* dependencies to be included in the graph, to cater for tasks which do involve side effects, and thus introduce hidden dependencies relating to other tasks which access the same state. An example of a task dependency graph is given in Figure 2.2.

Many workflow languages support conditional branching and iteration. Conditional branches are implemented using *distributor* and *selector* nodes [80]. A distributor node takes a control value and a data value, and uses the control value to determine on which of its two outputs the data value should be sent. A selector node does the reverse; it reads in a control value and determines which of its two inputs it should read a data value from, before passing it through on its single output. Iteration is achieved by adding a cycle to the graph, where an edge is connected from one task to another task which precedes it in the graph, and a conditional branch is used to determine when the loop should terminate. Other forms of iteration are often supported by extending the basic dataflow model with additional language constructs [11, 233]. Some workflow languages also support the definition of user-defined functions, often referred to as *sub-workflows*. A sub-workflow contains additional edges corresponding to its inputs and outputs, which are connected to other tasks by the caller.

While all dataflow models support task parallelism, only some support data parallelism. In *static dataflow* models [86], the structure of a graph remains fixed, and only one instance may be active at a time. For each node in the graph, the input values are associated with that node, and there is no additional information attached to the values. This model supports neither recursion nor data parallelism, both of which rely on each operation being used in multiple active function calls or loop iterations at a time. In contrast, *dynamic dataflow* models permit multiple instances of a graph to be active simultaneously. Hardware implementations of dataflow languages typically use a *tagged token* architecture to achieve this [130, 22], though an alternative scheme more useful for software-based implementation is to dynamically expand the graph at runtime, in a similar manner to graph reduction (discussed in Section 2.4.5).

It is typical for a workflow language to have both a visual and a textual syntax. Visual syntaxes show the nodes and edges in a similar manner to Figure 2.2, and are edited using a graphical user interface. The motivation for visual programming environments is that they are sometimes considered easier to use for those with little programming experience. Workflow graphs can alternatively be represented textually by writing a series of statements specifying the nodes, and another set of statements defining the edges, an example of which is given in Section 3.6.4. Many workflow languages use XML-based formats for their textual syntax. The visual and textual syntaxes provided by a given workflow system are equivalent in expressiveness, since they are simply different ways of representing the same information. The choice of syntax is orthogonal to the execution mechanism, and in fact some workflow systems implement editing and execution using different pieces of software.

Workflow languages are designed for *coordination* rather than *computation*, and provide poor support for implementing the sort of low-level code that one would typically write in general-purpose programming languages like C or Java. Even though control flow structures such as conditional branching, iteration, and sometimes recursion are provided by workflow languages, they are often considerably more awkward to express than in most other types of programming languages. We discuss the implications of these limitations in Section 2.3.8. Since it is sometimes necessary to perform computation within a workflow, most workflow languages allow the programmer to define tasks in separate scripting languages which can be used to express this computation.

2.3.4 Orchestration and choreography

There are two main ways in which workflow execution may be coordinated: *orchestration* and *choreography* [250, 33, 260]. The meaning of these terms is analogous to those in the performing arts. With orchestration, a single controlling entity directs a set of participants to do certain things. In the case of services, a single client manages execution of the workflow by performing a series of request/response interactions with each of the services. This is similar to the role of a conductor in an orchestra. With choreography, the services interact directly with each other, without any centralised control. This is similar to a group of dancers who have learnt a set of moves, and carry them out during a performance according to the way in which they have been trained.

Choreography is mostly used in business applications, as an abstract definition of the observable interactions between business partners [253]. The actual internal computations that each carries out are not the concern of choreography, which only addresses the interactions. When applying this concept to scientific workflows, this corresponds to each task having an internal implementation that is opaque to the outside world, and data being passed directly from producers to consumers [31]. Since scientific workflows sometimes involve large amounts of data, this can reduce the total amount of network bandwidth consumed by a workflow [29, 269, 32]. Choreography is usually specified explicitly [199, 326, 253], though in some cases it is possible to automatically decompose a workflow and choreograph it across multiple machines [237, 61], a topic we explore in Section 3.4.2.

2.3.5 Error handling

The distributed nature of workflow execution means that workflow systems must address the issue of error handling. Ideally, they should be fault tolerant — that is, if a task fails, the workflow engine should handle it gracefully. For transient failures such as network timeouts, it is often possible to re-execute a task on a different machine [242, 211]; the side effect free nature of most tasks used in scientific workflows makes it possible to do this safely without the risk of affecting the final result. For permanent failures such as tasks reporting invalid input data, the workflow system should make it easy for the user to find out what happened and determine how to correct the problem. Some systems allow a partially executed workflow to be restarted from the point at which it failed [296], so that it is not necessary to execute all of the other tasks again. Another approach is to incorporate exception handling mechanisms into the workflow language, so that the programmer can specify how to respond to errors [298].

While important, error handling and fault tolerance are outside the scope of our own work, as our focus is on programming models and implementation techniques. Nonetheless, our workflow model described in Chapter 3 leaves open the possibility of a fault-tolerant implementation. We discuss possible enhancements to our implementation to cater for fault tolerance in Section 8.4.3.

2.3.6 Execution management and monitoring

Workflow systems often provide facilities to manage, track, and monitor computations. At minimum, there must be some way for the user to start execution of a workflow and be notified of its completion, but sometimes there is additional functionality provided. Some workflow engines display a list of tasks to the user, indicating which ones are pending, active, or completed [242, 27]. Various performance monitoring architectures have been proposed to record events of interest on execution nodes, which can be used to report performance information to users or make scheduling decisions [25, 52, 304]. Provenance information, which indicates the source inputs and computational steps used to produce a data item, is also recorded by some systems [272, 273, 10]. A comprehensive coverage of issues surrounding execution management services for distributed applications is given by Foster et. al. [105]. These issues are outside the scope of this thesis, as our focus is specifically on programming models and related implementation techniques.

2.3.7 Languages and systems

Many different workflow languages have been developed over the years. State-oriented business workflow languages include BPEL [16], WSFL [198], XLANG [297], WSCI [20], BPML [19], and YAWL [311]. Data-oriented scientific workflow systems include DAGMan [76], Taverna [242], Kepler [210], Triana [294], JOpera [249], Karajan [317], GridAnt [316], PAGIS [323], ICENI [222], WildFire/GEL [291, 201], GridFlow [58], SCIRun [245], Gridbus Workflow [328], Pegasus [84], BioExtract [212], ScyFlow [220], and SWFL [143]. We are interested solely in data-oriented, scientific workflow systems, and in this section we study a representative selection of these. Our aim here is to give an understanding of the nature of these systems, as well as their capabilities and limitations. We primarily discuss the programming models provided, the implications of which are discussed in Section 2.3.8.

Unfortunately, existing workflow literature is sorely lacking in two crucial areas: implementation details and performance results. Papers on workflow systems tend to focus solely on the features of a given system, without providing a detailed explanation of the execution mechanism or an evaluation of performance and scalability. This makes it difficult to compare existing systems with our own in terms of these aspects. With a few exceptions [137, 115, 230, 32], we have found very little in the way of detailed performance evaluation, despite the fact that this is common for parallel programming languages [40, 206, 107], message passing libraries [204, 194], and other types of distributed computing systems [83, 94]. For this reason, the discussion in this section is limited to features and programming models.

2.3.7.1 DAGMan

The Condor system [296] described in Section 2.1 includes a tool called DAGMan [76], which can be used to define workflows in which tasks correspond to jobs submitted through Condor's batch queuing system. Each task is an executable program that is run on a worker machine, and may be written in any language. Input and output data is exchanged between tasks via files, which may either be stored on a shared file system, or transferred as part of the workflow using Stork [190].

DAGMan's programming model is based on directed acyclic graphs. The syntax specifies a list of tasks, each defined in a separate job description file, and a list of parent/child relationships indicating the order of execution that must be respected during execution. When the workflow is run, any set of tasks whose parents have all completed may execute in parallel. Pre and post scripts may also be specified for each task, which run before and after the actual task.

Control constructs such as conditionals, loops, and function calls are not supported. However, nested workflows may be implemented by having tasks which run a script or executable program that generates a DAGMan input file, and then runs DAGMan recursively. This form of usage was not considered as part of the design or implementation of DAGMan, but rather is a technique that was subsequently discovered by users. It has been successfully used to implement data parallel sub-workflows based on an unknown number of inputs.

DAGMan's target use cases are HPC (high performance computing) environments involving clusters or distributed networks of workstations, and thus it does not support services. It instead presents a model more suitable for situations where users have custom-written code or other applications that they wish to execute on remote resources, rather than being restricted to services that have already been deployed on other machines. This makes it somewhat more flexible than the service-based workflow systems described in the following sections, in that it allows user-supplied code to be executed on remote machines.

2.3.7.2 Taverna

Taverna [242] is a workflow system designed for performing *in-silico* (computation-based) scientific experiments. Although primarily intended for bioinformatics, most of its features are of a general nature that could be used in a wide variety of application domains. Taverna consists of both a workflow engine, *FreeFluo*, and a graphical editing environment, the *Taverna Workbench*. The workflow language implemented by Taverna is called SCUFL (Simple Conceptual Unified Flow Language), the formal semantics of which are given in [306].

Tasks are called *processors*, and there are many different types available. These include web services, local Java components, SQL queries, filesystem access, XML transformation, and code snippets written in the Java-like scripting language BeanShell [238]. A processor can also be a *sub-workflow*, allowing for nested specification of workflows, although recursion is not supported. Each processor has a set of input and output ports, and data links connect these to define the transfer of data from one processor to another. Task parallelism is supported; multiple tasks can execute in parallel whenever there are no data dependencies between them.

Conditional branching is achieved by adding tasks to the graph which are instructed to fail for a specific input value. When a task fails, it produces no output, preventing other parts of the graph that depend on it from running. A pair of "fail if true" and "fail if false"

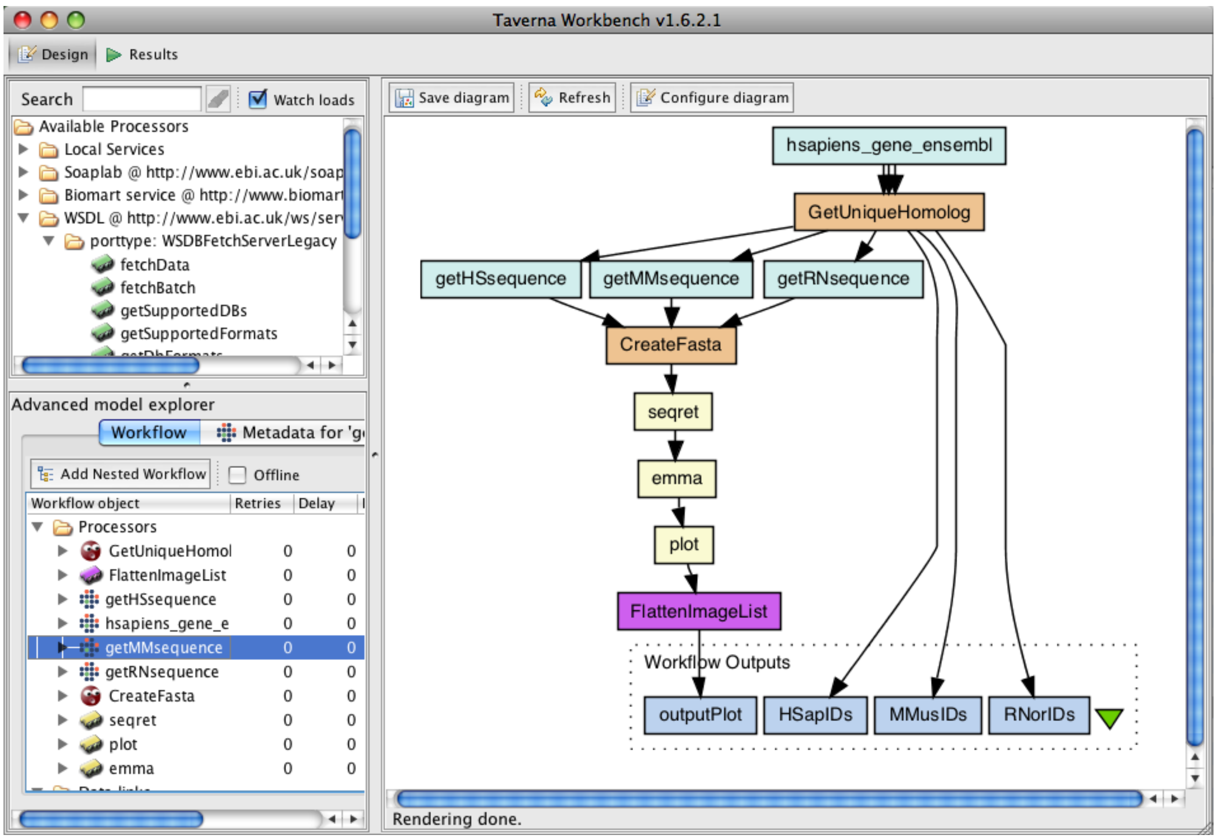


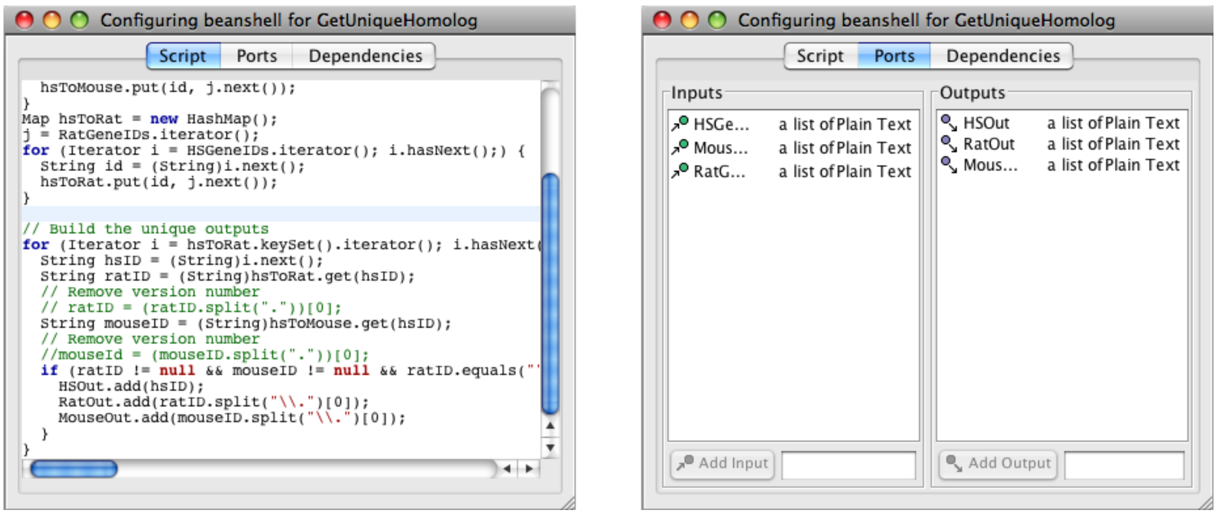
Figure 2.3: The Taverna workbench

tasks can thus be used to select between two different branches of the workflow, based on a boolean value supplied as input to both of them. Since Taverna lacks primitive operations like arithmetic and string comparison, any non-trivial conditions must be written in another language, such as BeanShell, and exposed as a task.

Taverna supports three different types of iteration. The first type is *implicit* iteration, where a task that is supplied with a list of items as input is executed once for each item in the list. If multiple lists are supplied, then either a *cross product* or *dot product* iteration strategy must be used. Cross product executes the task once for every possible pairing of items from the lists, and dot product executes the task for each pair of items that are in the same position in both lists. The implicit iteration scheme supports data parallelism, whereby a user-specified number of threads concurrently execute instances of the task for different list items [289]. The second type of iteration is a looping facility, which repeatedly executes an individual task until some condition becomes false. This has the significant restriction that the condition may only be based on the outputs of that particular task, preventing, for example, the comparison of the task's output with a value obtained from a portion of the graph that is not part of the loop. The third type of iteration involves adding cycles to the graph [118]; in this case, a conditional branch must be used for the termination condition.

Figure 2.3 shows a screenshot of the Taverna workbench. In the top-left corner, a list of known processors is available, including a large collection of web services obtained by querying service registries. Users may browse through this and select processors to add to the workflow. In the bottom-left part of the window, the structure of the workflow is shown as a tree. Editing takes place by dragging processors from the top-left palette to the tree structure, and configuring the data links and parameters for each processor. The graphical view of the workflow on the right shows the structure as a graph, with colours and labels indicating the types and names of the processors. Only limited editing support is possible using the graphical view; most editing operations require the user to interact with the tree view.

When the workflow is run, the user is presented with progress information showing which of the tasks are executing or have completed, as well as intermediate results that are generated. This is very helpful for interactive usage, especially when workflows take a long time to run, and there is a possibility of some tasks failing due to network problems or errors on the server (in which case retries occur). The user can view the intermediate and final results once execution is complete; provenance information is also recorded, so that it is possible to keep track of how data items are produced.


Configuring beanshell for GetUniqueHomolog
Script
Ports
Dependencies
Configuring beanshell for GetUniqueHomolog
Script
Ports
Dependencies
Add Input
Add Output

```
  hsToMouse.put(id, j.next());
}
Map hsToRat = new HashMap();
j = RatGeneIDs.iterator();
for (Iterator i = HSGeneIDs.iterator(); i.hasNext();) {
  String id = (String)i.next();
  hsToRat.put(id, j.next());
}

// Build the unique outputs
for (Iterator i = hsToRat.keySet().iterator(); i.hasNext(
  String hsID = (String)i.next();
  String ratID = (String)hsToRat.get(hsID);
  // Remove version number
  // ratID = (ratID.split("."))[0];
  String mouseID = (String)hsToMouse.get(hsID);
  // Remove version number
  //mouseId = (mouseID.split("."))[0];
  if (ratID != null && mouseID != null && ratID.equals("'
    HSOut.add(hsID);
    RatOut.add(ratID.split("\\.")[0]);
    MouseOut.add(mouseID.split("\\.")[0]);
  }
}
```

Inputs

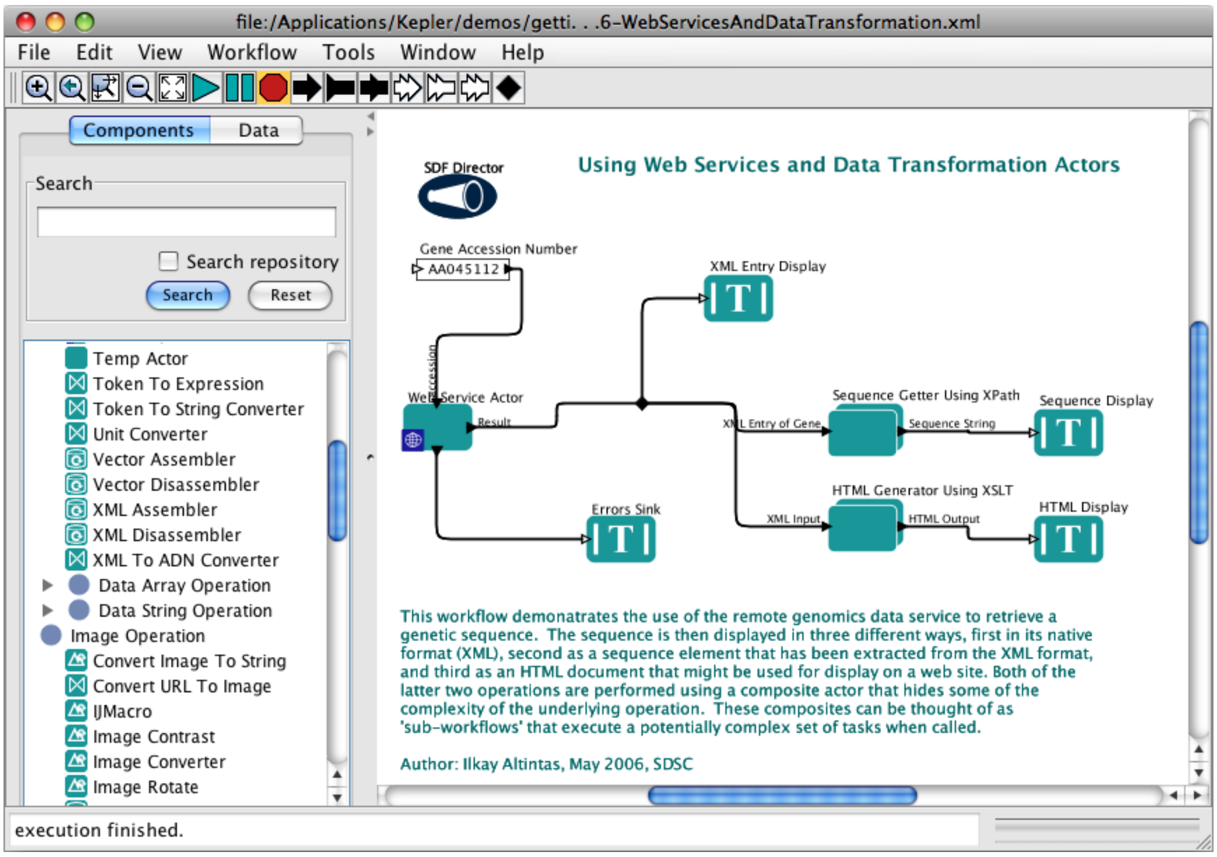
HSGe...	a list of Plain Text
Mous...	a list of Plain Text
RatG...	a list of Plain Text

Outputs

HSOut	a list of Plain Text
RatOut	a list of Plain Text
Mous...	a list of Plain Text

Figure 2.4: BeanShell process editor

If any parts of the workflow logic cannot be expressed in SCUFL, then a BeanShell task can be added to implement the logic. The script editor is shown in Figure 2.4, displaying the code and ports associated with the `GetUniqueHomolog` task from Figure 2.3. The user specifies the input and output ports of the task, and these are exposed to the script as variables which can be read from or written to. These embedded scripts are necessary if one wants to perform basic operations like arithmetic or data structure manipulation. This has the implication that in practice, users really need to have programming expertise in order to do anything non-trivial.



This workflow demonatrates the use of the remote genomics data service to retrieve a genetic sequence. The sequence is then displayed in three different ways, first in its native format (XML), second as a sequence element that has been extracted from the XML format, and third as an HTML document that might be used for display on a web site. Both of the latter two operations are performed using a composite actor that hides some of the complexity of the underlying operation. These composites can be thought of as 'sub-workflows' that execute a potentially complex set of tasks when called.

Author: Ilkay Altintas, May 2006, SDSC

Figure 2.5: Kepler

2.3.7.3 Kepler

Kepler [210] is another graphical workflow environment designed for e-Science applications. It is based on a similar model to Taverna, whereby a user edits a dataflow graph by dragging a set of tasks (called *actors* in Kepler) on to a canvas and adding links between input and output ports. Actors come in many different forms, including Java components, web services, file transfers, remote job submission, and XSLT transformations, as well as primitive operations such as arithmetic and string manipulation. Figure 2.5 shows a screenshot of Kepler, editing a simple workflow involving a call to a web service and subsequent processing of the data using embedded XSLT and XPath actors.

Kepler is based on Ptolemy II [153], a system designed for modeling and simulation of concurrent systems. Ptolemy II supports multiple models of computation, and separates the execution logic from the syntactic representation of a workflow through the use of *directors*, which play a role similar to that of programming language interpreters. The models of computation supported by Kepler of relevance to our study are *synchronous dataflow* [38], *dynamic dataflow* [337], and *process networks* [119]. All of these models have in common the property that the workflow as a whole does not contain any global state, and the only transfer of data that occurs is between actors that are explicitly connected in the graph. Of the three, the process networks director is the only one which supports parallel execution; the synchronous dataflow and dynamic dataflow directors both operate sequentially.

The synchronous dataflow director performs static scheduling, which involves determining the complete execution path of the workflow before execution begins. This prevents conditional branching and data-dependent iteration, both of which require decisions about which actors to execute to be made at runtime. Loops with a fixed number of iterations are supported, but these are of limited usefulness, since most algorithms involving iteration need to cater for different input sizes. The dynamic dataflow director performs dynamic scheduling, whereby it determines which actors to execute at runtime based on the availability of data tokens on input ports. This is similar to the scheduling approaches used by DAGMan and Taverna. Dynamic dataflow supports conditional branching, as well as both fixed and data-dependent iteration.

The process network director implements parallel execution by creating a Java thread for each actor in the workflow graph. Task parallelism is thus achieved, since multiple actors which all have input data available can execute concurrently. However, the approach of one thread per task rules out the possibility of data parallelism, since each task can only process one input at a time. It also has the downside of imposing significant overhead for fine-grained computation, since even simple actors like addition and string concatenation operators each require their own thread. Conditional branching and iteration are both supported by the process network director.

Based on our experience with Kepler, there does not appear to be an way to define recursive sub-workflows. Zhou [337] gives an example of a recursive function defined in Ptolemy II, which works by dynamically expanding the graph at runtime for each recursive call. However, this example relies on an actor that is not available in Kepler. It also appears to be specific to the dynamic dataflow domain; it is not clear whether this can be used within the context of a process network.

Processing of data collections is possible due to the fact that Kepler executes workflows in a streaming manner, whereby each data link may carry multiple tokens, with actors being fired once for each token they receive. This enables a given function (possibly a sub-workflow) to be applied to each item in a sequence, in a similar manner to `map` in functional programming languages, and the iteration capability of Taverna. Built-in actors are also provided for converting between token sequences and arrays. Other higher-order constructs for list processing are missing, resulting in some types of workflows that can easily be expressed in functional languages being awkward to construct in Kepler [209]. For the case of nested data collections, work by McPhillips, Bowers, and Ludascher [225] has improved the ability to deal with nested sequences by flattening them into sequences containing open/close delimiters, enabling streaming processing of nested data [225].

Recent work by Abramson, Enticott, and Altinas [4] added support for data parallelism to Kepler by implementing a new director based on the tagged token architecture originally developed for dataflow machines [130, 22]. This director operates by associating a colour with each token, and firing an actor whenever all its inputs have tokens available of the same colour. Parallelism occurs whenever there are multiple sets of tokens on an actor's inputs, with each set having a different colour. A new thread is spawned for each parallel

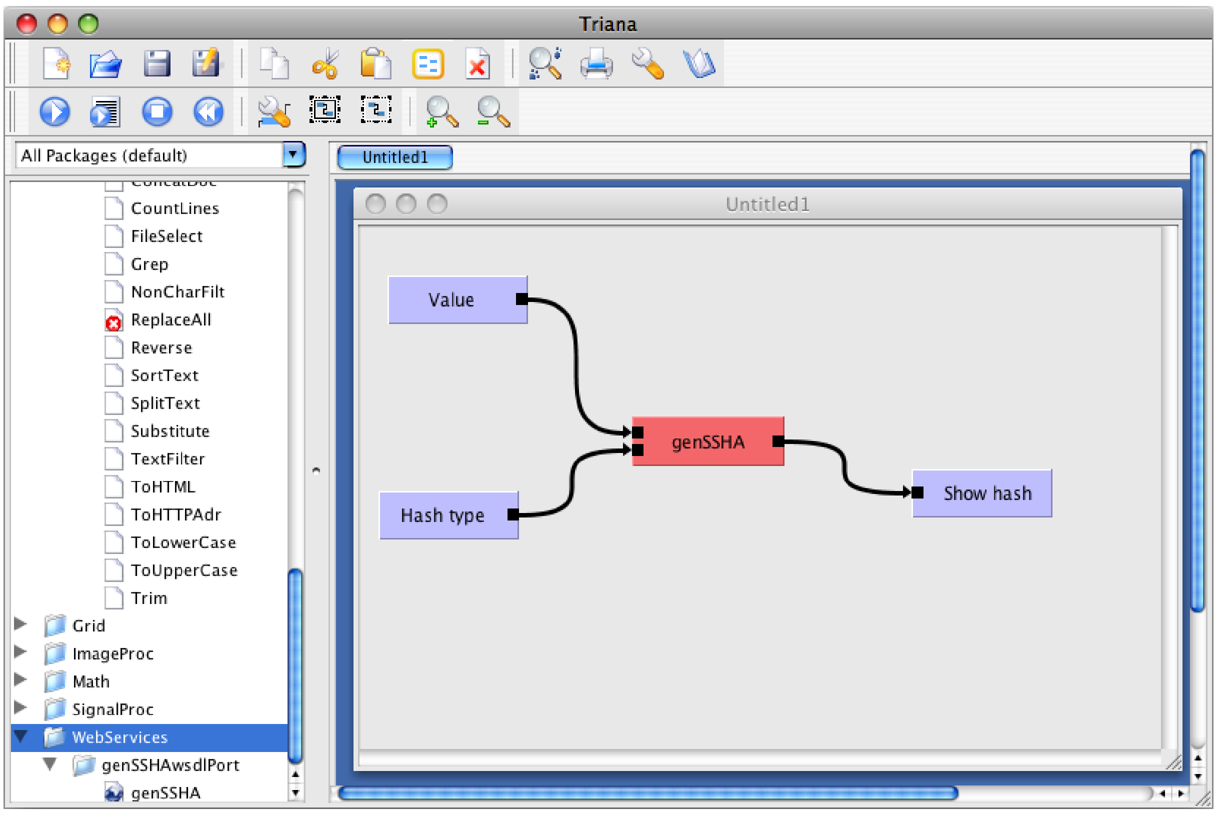


Figure 2.6: Triana

instance of the actor, each of which processes tokens of a given colour. Generation of tokens with different colours is achieved using actors specifically designed for introducing data parallelism. An example is the parameter sweep actor, which produces a stream of tokens containing values within a specified range.

2.3.7.4 Triana

Triana [70, 294] follows the same basic model of Taverna and Kepler, providing a graphical interface for editing workflow graphs. As with the two systems mentioned above, it contains a tool palette from which a collection of built-in tasks is provided which can be used as nodes in the graph. After constructing a workflow, the user may run it from the editor and view progress as a series of highlighted nodes showing the progress of execution. A screenshot of the interface is shown in Figure 2.6, depicting a simple workflow involving a call to a web service with two string constants passed in as parameters.

Job submission and service-based external tasks are both supported. In the former, a task corresponds to an executable that is run on a remote machine via a batch queuing system. Input and output of the task occurs via files, which are staged in before the program runs, and staged out afterwards. Within the workflow graph, the transfer between internal data values and files is handled by file nodes that are placed between producers/consumers of data and the tasks that use them.

Services are supported as another type of task, which can correspond to web services, or services implemented using other RPC protocols. In the former case, a UDDI registry [239] may be queried to locate services matching a particular search string, which are then added to the tool palette. As with other types of tasks, these are added by dragging them onto the canvas and connecting links up; these links correspond to the input parameters and return value of the service.

Support for complex data structures exchanged with web services is limited. A task can be added to the graph in which the fields of the structure can be explicitly specified as constants, but the documentation suggests that this is only useful for testing, and that custom Java classes should be written if structured data types need to be created dynamically during execution. This makes it difficult to work with services that use anything other than atomic types for their parameters and return values.

Basic control constructs, including conditionals and loops, are supported. In the case of loops, a dialog is provided to configure simple loop iterations and conditions. More complex loop conditions must be implemented by writing a separate Java class to compute the condition, and associating that class with the loop node. Loops are executed using sequential iteration; data parallelism is not supported. Sub-workflows can also be defined, and these can be used within loops as well as separately, just like other tasks.

A distributed execution mode is provided whereby a set of machines running Triana may participate in execution of a workflow. The graph can be split up by having the user explicitly select groups of nodes which are to be exposed as a service, which can then be invoked from instances of the Triana workflow engine running on other machines. This can also be done with web services, enabling all or part of a workflow to be invoked from clients written in other languages.

In addition to job submission and services, a number of other built-in tasks are available, and custom ones can be written in Java. Some of the provided tasks include generic operations like string manipulation and arithmetic, as well as domain-specific tasks like those for image and signal processing. These domain-specific tasks reflect Triana's focus on being a problem-solving environment useful for scientists in particular disciplines, as well as a more generic tool for service composition and distributed job management.

2.3.7.5 JOpera

JOpera [249, 247] is a workflow language designed for composing web services. It features a graphical editor which allows users to construct the workflow in a visual manner similar to the other systems discussed previously. A workflow in JOpera is represented by two different graphs which must be edited separately: one containing control dependencies, and another containing data dependencies. Any edges added to the dataflow graph are automatically replicated in the control flow graph, since data dependence implies a constraint on execution ordering. Additional control dependencies may be added if tasks containing side effects are used within the workflow.

Task parallelism is supported in the usual manner, by analysing dependencies in the graph. In contrast to Kepler's process networks director, JOpera decouples the thread management from the actual tasks. Instead of having a single thread for each task, a set of dispatcher threads is maintained which are able to execute any tasks that are in an executable state. Service requests are made using asynchronous I/O operations, so that a thread does not have to remain blocked while waiting for a response. This technique provides greater scalability to that of Kepler's, since the number of outstanding service requests and tasks in a workflow is not limited by the number of threads that can run on a single machine. JOpera also supports a distributed execution mode where workflow coordination logic and tasks can be distributed across multiple machines in a cluster [137].

In addition to web services, other types of tasks may be used in a workflow, including command-line programs, SQL queries, XSLT transforms, and job submissions to batch queuing systems. For fine-grained computation which cannot be expressed within the workflow language itself, *Java snippet* tasks may be added in which the user can write arbitrary Java code, in a similar manner to BeanShell scripts in Taverna. Sub-workflows (user-defined functions) are also supported; these have the ability to call themselves recursively.

Conditionals are handled by *start conditions*, which are associated with every task in the graph. A task whose inputs are all available will only begin execution if its start condition evaluates to true (the default), otherwise it will be skipped. The equivalent of an `if` statement can thus be modeled by having two sub-graphs whose root nodes have inverse start conditions, and passing the values used by those start conditions as input to both nodes. There is no explicit conditional task type, since conditions can be specified on any task. When two alternative execution paths both provide an input value to the same task, the task has a data dependency on both, but will fire as soon as a value arrives from either of the edges.

Two forms of iteration are supported. The first form is control flow loops, whereby cycles are added to the graph to enable tasks to fire repeatedly until a start condition evaluates to false. This type of iteration happens sequentially. The second form of iteration is so-called *list-based* loops, which execute a given sub-graph once for each item in a list, in the same manner as `map` in functional programming languages. This mechanism supports data parallelism by dynamically expanding the graph when the loop begins; a copy of the sub-graph is made for every list element, all of which can execute in parallel. The way in which these types of loops are specified is through the use of *split* and *merge* nodes (which would more appropriately be called *scatter* and *gather*); a split node takes a list and outputs one value for each list item, and the merge node does the reverse.

Basic support is provided for manipulating XML data structures exchanged with web services: For every complex type defined by a web service interface, JOpera automatically generates a *pack* and *unpack* operation. The pack operation takes one parameter for each field, and produces a compound data structure containing all of the fields; unpack does the reverse. For lists of XML elements, the split and merge operators used for list-based loops can be used to iterate over the contents of the lists. If more complex XML manipulation is required, the user can insert an XSLT task in the workflow, in which they can write code to perform arbitrary transformations.

2.3.7.6 PAGIS

PAGIS [323] is a workflow system based on a process network model similar to that supported by Kepler. It provides both a graphical user interface and a Java API for constructing process networks, where each process is implemented as a Java class, and runs in a separate thread. Distribution is implemented using a metaobject protocol [185], which allows method calls to be intercepted and redirected to other machines. Processes can be migrated between machines in order to balance load.

A significant feature of PAGIS is that it allows programmatic reconfiguration of the process network at runtime [322]. The metaobject protocol provides reflective APIs which allow a process to manipulate the workflow graph in various ways, such as adding or removing other processes. To ensure consistency of the graph, modifications are queued in a *builder* object, and then applied atomically as a complete set. Compared to a static graph model, this dynamic reconfiguration has the advantage of enabling more flexible control flow structures to be implemented. However, doing so requires the programmer to express this control logic in Java, instead of a higher-level workflow language.

2.3.7.7 Karajan

Karajan [317] is a workflow language that is much closer to the notion of a traditional scripting language than the others mentioned here. It was constructed after experience with an earlier workflow language, GridAnt [316], which utilised simple task dependency graphs. Karajan is aimed primarily at coordinating large HPC workflows involving submission of jobs to batch queuing systems on remote compute nodes. The external tasks it supports mainly consist of launching jobs and transferring files from one machine to another. However, the general design is extensible, and custom tasks can be implemented in Java to perform any operation that a programmer wants.

The programming model is essentially imperative, though several constructs are provided for explicitly specifying parallelism. These include fork/join style parallelism, in which several statements are executed concurrently, as well as parallel iteration (similar to `map` in functional programming languages). There is also support for a reasonably flexible collection of regular control-flow constructs such as conditionals and loops, as well as user-defined functions.

Limited forms of data manipulation are possible. Variables and function arguments may be atomic values or lists, and the latter can be nested to create arbitrary data structures. Karajan uses dynamic typing, and provides a small collection of built-in functions for dealing with data values, such as list manipulation, string operations, and basic arithmetic. A hash table structure is also supported, with the standard key/value-based operations for inserting and retrieving entries. This support for working with data distinguishes Karajan from most other workflow languages, which normally treat data as opaque. However, the facilities provided are not as sophisticated as those provided in mainstream scripting languages.

Two different syntaxes are supported, with identical semantics. One is based on XML, in which each language construct is expressed as an element, with properties specified as attributes, and child nodes of the syntax tree are written as child elements. This syntax is similar to that used by languages like XSLT [179] and BPEL [16]. A more programmer-friendly syntax is also provided, where each language construct is written in the form `f(arg1,arg2,...)`. This is particularly well-suited to larger scripts, and is much closer to what most programmers are familiar with.

The execution engine supports a distributed mode in which a Karajan service runs on a remote host, and a workflow may explicitly request certain portions of its code to be evaluated by the remote node. This has some similarities with our distributed execution mode described in Section 4.7, but requires the programmer to explicitly direct the distribution, whereas our implementation does this implicitly.

One of the main limitations of Karajan is that since it is designed exclusively for job submission-based workflows, it does not support web services or any other type of generic RPC protocol. Accordingly, there is no way to express the types of service-based workflows that one could create in Taverna, Kepler, Triana, or JOpera. Based on our inspection of Karajan's source code, it appears that it would be practical to extend it with this support; doing so would likely require modifications to the type system and the way in which functions are made accessible, as well as the addition of XML processing features. Another limitation is that since Karajan is primarily an imperative language, parallelism must be specified explicitly by the programmer, in contrast to the dataflow-style languages described earlier.

2.3.8 Limitations

The workflow languages described above are sufficient for simple workflows, in which all of the computation is carried out by external tasks. However, many real-world workflows involve some degree of internal computation, such as small calculations on data values passed between services, and conversion of data between different formats. When such functionality is needed, the developer is forced to switch to a separate, more expressive language in order to implement additional tasks to be included in the workflow. The overly simplistic design of workflow languages can thus sometimes make it *harder* to implement a workflow than if a more sophisticated and flexible language design had been provided in the first place.

In terms of control structures, the limited iteration support provided by some workflow languages is insufficient to express some types of data parallelism, in which multiple elements of a list or data structure need to be processed independently in parallel [114, 4]. Even within a dependency graph, iteration must often happen sequentially, as it is expressed using a cycle in the graph [209]. Conditional statements are often awkward to express, particularly in the case where a complex expression involving the outputs of several tasks needs to be evaluated in order to choose a branch. Recursion within user-defined functions (sub-workflows) is also often difficult or impossible to express. Programmers familiar with general-purpose programming languages are likely to find the way in which these sorts of features are implemented to be cumbersome to work with.

With regards to data manipulation, a particularly common requirement is conversion between data formats used by different services [321, 34]. When constructing a workflow involving syntactically incompatible services, additional tasks must be added to convert between the different formats; these tasks are known as *adapters* [254] or *shims* [152, 151]. Because the workflow language does not contain the features necessary to express the data conversion, these shims must be implemented in a different programming language [200]. Oinn et. al. [242] discuss the development of *shim libraries*, which deal with known types and can be re-used in different workflows. An idea that builds on this is to automatically determine appropriate shims based on type information [46, 150, 288]. However, these approaches are fundamentally limited as they do not cater for custom, application-specific formats or other operations such extracting individual fields of a data structure.

An alternative solution is to incorporate user-programmable task nodes into the workflow graph. These tasks contain a small program written in a scripting language, which is invoked by the workflow to perform the conversion. Such task nodes can in fact be used not just for data conversion, but also for arbitrary computation, enabling local application logic to be included in the workflow. Examples of these approaches include Kepler's XSLT and XPath actors [210], Taverna's support for scripts written in BeanShell [238] and XQuery [279], and JOpera's Java snippets. In the context of state-oriented business workflows, a proposed extension to BPEL called BPELJ [42] adds the ability to include snippets of Java code within a workflow.

The existence of these extensions clearly indicates a demand for more expressiveness in workflow languages. We can look to the history of scripting languages for guidance here. Early scripting languages were designed simply to execute a sequence of shell commands instead of typing them in manually. Over time, features such as variables, control flow constructs, and simple expression evaluation were added, enabling a wider range of logic to be implemented in scripts. Eventually, languages like Perl and Python emerged, possessing the full range of expressiveness found in systems programming languages like C. Workflow languages can be considered the *distributed* equivalent of scripting languages, and we believe in incorporating increased expressiveness into them in a similar manner. In achieving this, it is necessary to cater for the unique features of workflow languages, notably automatic parallelisation and integrated service access.

Fortunately, there is already a large body of work we can look to for answers. The side effect free, data-oriented nature of scientific workflows, along with the requirement for parallel execution, ties in very closely with *parallel functional programming languages*. Past research on these languages has resulted in the development of implementation techniques that can handle parallelism well and efficiently execute fine-grained computation. And because functional languages are based on the underlying model of lambda calculus, they are capable of expressing arbitrarily complex computation — solving the problem of expressiveness. In Section 2.4, we give an overview of this work and discuss how it relates to workflows. Existing parallel functional languages do not entirely meet all of our needs however, and in Chapters 3 and 4, we describe the work we have done to incorporate ideas from both areas.

2.4 Parallel functional programming

Functional programming [117, 144] is a style of programming that emphasises computation of results, rather than manipulation of state. It is based on the concept of a function in the mathematical sense — that is, a mapping from one or more input values to an output value. All functions and expressions within a program are side effect free, a property which makes such programs easier to formally reason about. The lack of side effects also makes parallelism much more straightforward to implement than in imperative languages.

The property of being side effect free is something that is shared with the data-oriented workflow languages described in Section 2.3. These serve the purpose of computing a result, rather than manipulating state. All tasks are side effect free, and the input/output relationships and execution ordering between the tasks is expressed explicitly in the workflow structure. Functional programming languages are normally intended for programs in which all computation is done directly within the program itself, rather than being delegated to external services, but the basic idea is the same as in data-oriented workflows.

The underlying theoretical model behind functional programming is *lambda calculus* [67, 231]. It defines an abstract expression syntax as well as conversion operations which can be used to derive the the result of an expression. The model is extremely simple, yet capable of expressing arbitrary computation, since it is Turing complete. A brief tutorial on lambda calculus and an evaluation technique known as *graph reduction* is given in Appendix A.

An important property of lambda calculus and, by extension, pure functional languages, is the Church-Rosser Theorem [69]. This states that the result of evaluating an expression is independent of the order in which the individual reductions are performed. An implication of this theorem is that independent reductions can be performed in parallel, without the risk of introducing non-determinism. This is a key benefit of functional languages over imperative languages in terms of parallel programming, because it avoids the need for the error-prone synchronisation constructs necessary in multi-threaded or message passing programs. An overview of parallelism in functional programming languages is given in [133].

Functional languages provide the ability to define *higher-order* functions, which permit other functions to be passed in as parameters or returned as results. Higher-order functions contribute to the expressiveness of a language by allowing programmers to define a range of control structures in user code that would otherwise have to be natively supported by the language itself. This ability, inherited from lambda calculus, is an important tool for defining behavioural abstractions which lend themselves to a separation between the high-level logic of a program and its implementation details. For example, certain types of looping constructs and data structure traversal strategies can be defined in this manner.

Modern functional languages include many high-level constructs that would be awkward for programmers to express directly in lambda calculus. An early stage in the compilation of these languages involves translating source files into an intermediate language which is based on lambda calculus, and then performing further compilation steps from there [161]. Since lambda calculus is Turing complete, it is possible to implement certain classes of high-level language constructs by writing transformation rules which define how to express these constructs in terms of lambda calculus [170]. In Chapter 5, we use this approach to implement XQuery [43].

Our interest in functional programming within the context of this project is to use it as a model for defining and executing workflows. Its data-oriented nature sits well with that of scientific workflows that compute a result using side effect free operations combined using data dependencies. While much research has been done independently on both scientific workflow tools and functional programming languages, very little attention has been given to combining the two areas. Section 2.4.7 discusses the relationship between the two in more detail, and describes existing work that has touched on this idea.

Popular functional languages include Haskell [145], Miranda [309], ML [116], Common Lisp [280], Scheme [287], and Clean [53].

2.4.1 Modes of evaluation

Functional programming languages may use either *strict evaluation* or *lazy evaluation*. With strict evaluation, all of the arguments to a function are computed *before* the function call begins. Imperative languages are always strict. With lazy evaluation, arguments to a function are only evaluated *after* the call begins, and only if and when they are actually used. Such expressions are said to be *delayed*. The lack of side effects means that it does not matter when evaluation of an expression occurs, since a given expression may only produce one specific result.

Lazy evaluation enables certain types of processing to be expressed in a more abstract manner than is possible with strict languages. For example, it is possible to define data structures which are conceptually infinite, whereby only the finite portions of the data structure that are accessed by the program are actually evaluated. Stream-based processing is also a common use of lazy evaluation; for example, the output generated by a program may be obtained by traversing a lazily produced list whose elements are only evaluated once it is time to print them out. Lazy evaluation is also necessary for languages which treat `if` as a function, since, for example, evaluating both the true and false branches of an `if` statement within a recursive function would lead to non-termination. Cyclic data structures, like those used in Section 5.5.5, also require lazy evaluation in order to be constructed in a side effect free language.

Although useful, lazy evaluation has significant implications for performance [136]. While it can potentially reduce execution time by avoiding unnecessary computation, more often than not the overheads involved in memory allocation for delayed expressions actually harm performance. Every time an expression is passed as a parameter, a *closure* or *thunk*

[41] must be allocated to hold a reference to the function's code, as well as the values of all variables that it uses. There is also a cost involved with checking and forcing evaluation of these closures when their results are actually needed.

Most functional language implementations use a technique known as *strictness analysis* [73] to statically detect cases in which lazy evaluation can safely be avoided. For example, an expression involving the addition of two numbers definitely needs both of its arguments to be evaluated. The + function is said to be *strict* in both arguments, and strictness analysis will use such information from functions that the program calls in order to annotate expressions to indicate that they can be evaluated as soon as they are encountered. The compiler may then use these annotations to generate code that evaluates the expression directly, instead of allocating a thunk.

Strictness analysis is undecidable in general. The reason is that strictness annotations can only be safely added to an expression if it is known that the result of the expression will definitely be used. If there is a possibility that a particular expression may not be used by the program, then it could potentially correspond to a non-terminating computation. Due to the well-known halting problem [307], it is impossible to reliably detect non-termination in all cases, so strictness analysis must err on the side of caution, possibly missing some cases. A large body of work has been devoted to solving the problem in as many cases as possible [35, 267, 148].

2.4.2 Parallelism and efficiency

Theoretically, it is possible for a functional program to be automatically and efficiently parallelised across multiple processors or machines, without any explicit direction from the programmer. However, the task of achieving optimal or near-optimal parallelism has proven extremely difficult in practice, as straightforward techniques of parallelising a functional program can often introduce inefficiencies. Thus, while parallelism is easy to achieve, efficiency is not, so any implementation of a parallel functional language must be careful in how it manages parallelism [88].

A key issue for efficiency is granularity [207]. Each instance of an expression evaluated in parallel carries some overhead. If there are a large number of small expressions, the overhead will be much greater than for a small number of large expressions. It is thus better to group multiple expressions together wherever possible so that the overheads are minimised, but not to the extent that the parallelism in a program is reduced so much that it can no longer exploit all of the available processors. Virtually all existing work on parallel functional programming has focused on fine-grained computation, which has necessitated significant effort being invested to optimisation.

Parallelism within a functional language implementation may be either *implicit* or *explicit*. In the implicit case, there are no language constructs for specifying which expressions should be evaluated in parallel, so the compiler and runtime system must therefore make all such decisions. In the explicit case, the programmer annotates their code to indicate which expressions can be evaluated in parallel, and these act as hints to the runtime system as to when to *spark* expressions, marking them as candidates for parallel evaluation. Explicit annotations often permit the language implementation to achieve greater performance by avoiding too much sparking [55, 146], but come at the cost of increasing the work of the programmer.

Language designers are faced with tradeoffs that must be chosen based on the requirements of the language and expected properties of their target applications. The decision of whether to use implicit or explicit parallelism must be based on both the desired level of simplicity to be exposed to the programmer, as well as the granularity of the computations that are likely to be expressed in the language.

For languages designed primarily for fine-grained, CPU-intensive processing, the usability cost of requiring explicit annotations can be justified by performance gains; most existing parallel functional languages fit within this category. A consensus has been reached that for fine-grained computation, implicit parallelism is infeasible [166]. However, since most of the performance problems with automatic parallelisation are due to an excessive number of sparks being generated, programs that utilise smaller numbers of coarse-grained operations are less susceptible to these issues. This is true in the case of workflows, even if we take into account a moderate amount of computation being done within the workflow itself, since the majority of the program's execution time is still spent in external tasks.

Some functional languages do not include language constructs for annotating expressions with information about parallel evaluation; in order to achieve parallelism in these languages it is therefore necessary to rely on implicit parallelism. An important part of the work presented in this thesis is a parallel implementation of XQuery we have developed for use as a workflow language. XQuery does not include parallel annotations, and our implementation is intended for workflows involving coarse-grained operations. We therefore elected to use implicit parallelism, while accepting a degree of inefficiency for fine-grained computation.

2.4.3 Expressing parallelism

Successful approaches to efficiently parallelising fine-grained computation in functional languages rely on a degree of explicit direction from the programmer. These prevent the runtime system from generating too much parallelism, which can reduce performance due to the overheads of excessive sparking and thread creation. Explicit control also allows the programmer to utilise their knowledge of the program to provide better guidance about what should be parallelised than what could be inferred automatically using existing techniques. The fact that low-level details of thread creation and synchronisation are abstracted away means that the development task is still considerably easier than in imperative languages. A good survey of techniques is given in [303].

Glasgow Parallel Haskell (GpH) [302] extends the Haskell language with two functions, `par` and `seq`, for controlling parallelism and order of evaluation. The `par` function takes two arguments; it sparks its first argument and then returns its second. Sparking an expression marks it as a candidate for parallel evaluation, adding it to the spark pool which is consulted by idle processors when looking for work. The `seq` function evaluates its first argument, then its second, returning the latter. Together, these functions can be used to explicitly instruct the runtime system as to *how* a program should be executed. Neither changes the result of execution; both are simply annotations used for coordination purposes.

In an effort to avoid explicit usage of `par` and `seq` throughout programs written in GpH, Trinder et. al. [301] proposed *evaluation strategies* to make the parallelism more implicit. An evaluation strategy is a higher-order function which uses `par` and `seq` internally in order to achieve some particular evaluation behaviour, such as evaluating all elements of a list in parallel. By using evaluation strategies for common data processing patterns, programmers can keep the main application logic largely separated from the explicit specification of parallelism, though the program must still contain explicit annotations indicating the evaluation strategies to be used.

Data Parallel Haskell [62] uses ideas originally developed in Nesl [40], and applies them in a functional programming context. It automatically parallelises aggregate operations on nested arrays, by flattening the arrays in order to make them easier to partition and divide among threads. Compile-time transformations are applied to the code to *vectorise* it, so that the resulting sequential code can be executed by each thread over its own portion of the array. This implementation of data parallelism does not use annotations, but instead requires programmers to express their computation in terms of array types and data-parallel versions of common functions like `map` and `sum`.

These are just a few of the many techniques that have been used for parallel functional programming, and which are most closely related to our own work. Other techniques have been proposed which involve more explicit direction from the programmer, lessening the benefit of using a functional language. Examples of these include Eden [51] and Caliban [293]. A comparison of a range of different parallel functional languages is given in [206].

2.4.4 Laziness and parallelism

Lazy evaluation causes the evaluation of an expression to be delayed for as long as possible. An expression will only be reduced if and when its result is actually needed. This property has the effect of forcing almost all evaluations to happen in a sequential order. Parallelism in lazy languages can only be achieved effectively by utilising information indicating that an expression will be needed in the future, and can thus begin evaluation earlier. This information may come from explicit annotations added by the programmer, or from the results of strictness analysis performed by the compiler.

Because strictness is undecidable in the general case, and detection must therefore be conservative, solutions to the problem are very difficult to implement. Although the problem has been studied at length, existing techniques have not proven effective enough to catch all relevant cases of parallelism. This is part of the reason why manual annotations are often used instead, where the job of instructing the compiler which expressions can be evaluated in parallel falls to the programmer.

Results obtained from automatically parallelising lazy languages have been mixed. Kaser et. al. [174] achieved considerable success using a technique known as *ee-strictness analysis* [268], and good results were obtained for the programs it was tested with. However, this study only considered programs which did not actually rely on laziness as such, in the sense that the programs would still have behaved correctly under strict evaluation. Another investigation by Tremblay and Gao [299] looked into the problem in more depth, and concluded that while good results may be achievable for programs which do not need laziness, those that do are unable to have much parallelism extracted from them — although the authors do not state if the programs they tested which require laziness were inherently sequential or not. Klusik, Loogen, and Priebe [188], the designers of the Eden language, explain their deviation from lazy evaluation in certain cases in order to achieve a greater degree of parallelism. The unfortunate interaction between laziness and parallelism and the difficulty of implementing effective strictness analysis is one of the reasons behind the use of annotations by languages such as GpH [302]. Another investigation which concludes that annotations are necessary to achieve sufficient parallelism is described by Roe [259].

For list processing, it is possible to enable parallelism by providing a mechanism by which the programmer can explicitly demand the values of multiple list elements in advance of those items actually being used. This permits the consumer of a list to dictate the degree to which eager evaluation is performed. Martinaitis, Patten, and Wendelborn [218, 219] have applied this technique to a demand-driven model of streaming components, by carrying out the input demand and computation phases of a function call separately. However, their work is presented in the context of asynchronous method invocations in an object-oriented programming model, and these ideas have yet to be explored in the context of a functional programming model.

Some implementations of lazy functional languages use *speculative evaluation*, where expressions that are not yet known to be needed may be evaluated if enough resources are available, in order to save time if their results are eventually required. This significantly complicates runtime management, since it becomes necessary to prioritise necessary work over speculative work, and to terminate evaluation of the latter if and when it becomes known that the result will not be needed. While significant work has been done in this area [155, 88], we do not consider it in this project, since in the context of workflows, a client should avoid dispatching potentially unnecessary work to remote services shared with other users.

Another option of course is to not use laziness at all, and instead rely solely on strict evaluation. Some functional languages, such as Standard ML [116] and SISAL [100], are already strict, in which case the problems described above cannot occur. In other languages though, introducing strict evaluation would alter the language semantics, since certain types of programs may not terminate using the latter mode. Because fully strict evaluation is not an option for some languages, annotations and/or strictness analysis must be relied on instead. When creating a new language though, the designer has the luxury of taking this issue into consideration when deciding which evaluation mode to use.

One of the issues we explore in this thesis is the idea of a language implementation that supports either mode. By having a compile-time option to set the evaluation mode, it is possible to compare the performance of a program under both cases, and quantify the penalty that laziness incurs for a given program. Additionally, for programs which rely on laziness in some parts but not in others, we explore the use of a hybrid mode, in which strict evaluation is used by default, but can be opted out of where necessary. Section 6.1 discusses these issues further, and gives a performance comparison between the two modes.

In the next section, we discuss an evaluation technique called *graph reduction*, which is often used for implementing functional languages. The technique supports both strict and lazy evaluation, and has successfully been used in the past for both sequential and parallel language implementations. It is used as the basis of our virtual machine design, described in Chapter 4.

2.4.5 Graph reduction

Graph reduction is a common technique for implementing functional programming languages. It is based on the idea of representing the program as a graph, derived from its original syntax tree, and performing a sequence of reduction transformations on it to compute a result. As the dependencies are explicitly expressed in the graph, it is possible to reduce independent parts of it in parallel. Graph reduction has similarities with the models used by some of the workflow systems described in Section 2.3, but has traditionally been used for fine-grained computation. A brief description of how graph reduction works is given in Section A.4. Detailed guides explaining how to implement graph reduction can be found in [161] and [169]. A survey of approaches to graph reduction for strict and non-strict languages is given in [187] and [71].

Naïve implementations of graph reduction based on explicitly traversing the graph and performing individual reductions are slow, as the operations required to perform these transformations are expensive and involve a lot of memory allocation. To address this problem, most implementations utilise an abstract machine to encode the set of reduction operations necessary for a given program, which enables most of the work to be done without the expense of graph traversal. These abstract machines operate in a manner similar to bytecode-based language implementations such as the Java Virtual Machine, and may be implemented as either interpreters or compilers.

The first such abstract machine was the SKI-machine [308], which relied on a fixed set of combinators (functions) to perform reduction. Due to the single argument at a time style of function application, and the amount of temporary memory allocation required, this approach was quite inefficient. A major subsequent development was the G-Machine [157], which used *supercombinators* [149] derived from lambda lifting [158] to perform reduction more efficiently, applying multiple arguments at a time. The instruction set of the G-machine enabled the reduction operations to be encoded as a sequence of instructions, which could be compiled into machine code [159]. This work paved the way for numerous other designs based on the same basic ideas, such as the Spineless G-machine [54] and the Spineless Tagless G-Machine [164].

Due to the differences between the functional programming model and the state-based von Neumann architecture, much work was carried out in the 1980s on the design of specialised hardware architectures for executing functional languages. These contained multiple processors and shared memory, with different parts of the program being distributed among the processors. The memory, communication facilities, and instruction sets provided by these machines were tailored towards the requirements of parallel graph reduction, with the aim of obtaining better performance than could be achieved using mainstream hardware designed for sequential programming. Examples of such projects include ALICE [79], NORMA [263], MaRs [75, 60], and GRIP [72, 162].

Improvements in mainstream processor technologies made possible by the resources and financial investments of large processor manufacturers soon overtook these custom hardware projects. Due also to further developments in compiler technology for both functional and imperative languages, research into hardware support for functional languages was abandoned in favour of using commodity hardware. Some of the work that had been done on custom architectures made its way into software implementations of the same abstract machines. These utilised similar execution and load management techniques, but implemented them on top of general-purpose hardware and operating systems.

2.4.5.1 Parallel graph reduction

Implementations of parallel graph reduction fall into two major categories: *shared memory* and *distributed memory*. In both cases, each processor operates on the graph by independently reducing different expressions, and accessing other parts of the graph as necessary. Distributed implementations include what is effectively an object-based distributed shared memory system as part of their runtime system, which is tightly integrated with the load distribution and function call evaluation mechanisms. As is the case with all parallel programming technologies, shared memory systems can achieve greater performance for modest numbers of processors due to lower latencies, though distributed memory systems are more scalable.

Shared memory implementation is the simpler of the two approaches, because all processors have direct access to the graph stored in memory. Synchronisation is necessary between different threads that access the same portion of the graph, and this is typically achieved by associating locks with nodes in the graph. Suspension of threads occurs when one is in currently evaluating an expression that is needed by another, with resumption occurring when the expression has finished being evaluated. The synchronisation and blocking mechanisms are important issues to address to achieve acceptable performance. Load balancing is simpler than for the distributed case, since all processors have access to a single pool of work where they can deposit and retrieve sparks (expressions to be evaluated). Examples of shared memory implementations include Buckwheat [121], the <v,G>-machine [24], GAML [217], WYBERT [193], EQUALS [175], and GHC [135]. The architecture of shared memory implementations is depicted in Figure 2.7.

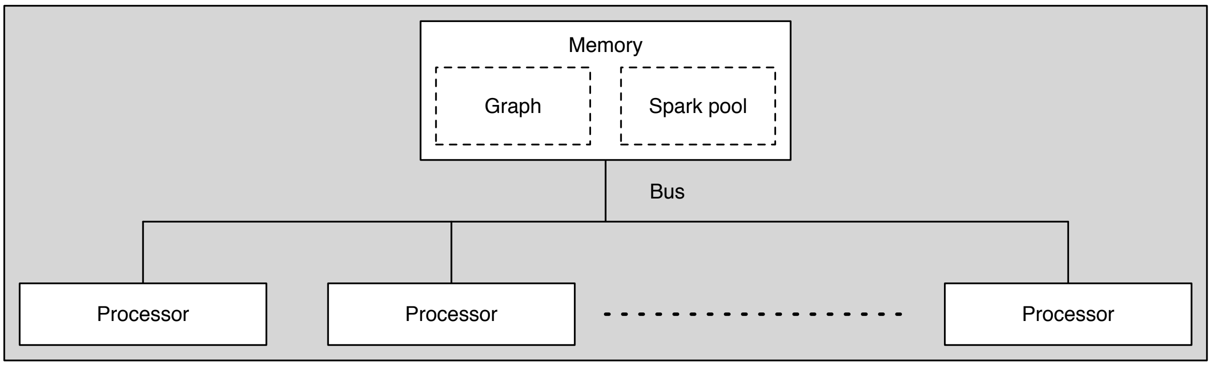


Figure 2.7: Shared memory parallel graph reduction

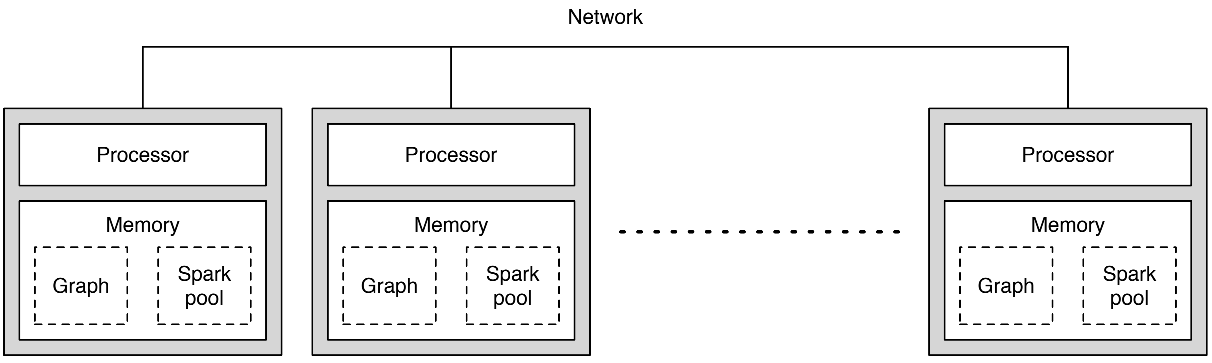


Figure 2.8: Distributed memory parallel graph reduction

Distributed memory systems are more complex to construct for several reasons. Firstly, the graph is stored across several machines, so mechanisms must be implemented to retrieve values remotely and manage suspension and resumption of threads that perform remote accesses; distributed garbage collection is also necessary. Secondly, a communication subsystem must be implemented to coordinate the tasks, though it is often sufficient to use existing message passing libraries such as MPI [229] or PVM [286]. Thirdly, the load distribution mechanism must operate using multiple spark pools, each of which resides on a different machine. Finally, the latency involved in communicating between processors must be taken into account to minimise the amount of time a processor is kept idle waiting to retrieve data. Distributed memory graph reduction implementations include the Dutch parallel reduction machine [26], Alfalfa [120], PAM [208], the HDG-machine [186], GUM [300], Naira [172, 171], and a JVM-based G-Machine [44]. The architecture of distributed memory implementations is depicted in Figure 2.8.

Most work on parallel functional programming was done prior to the Internet becoming mainstream, and all of the distributed memory implementations mentioned above are designed for clusters. However, recent work by Al Zain et. al. [335] has studied the case of wide-area distributed systems consisting of clusters at different sites, connected via the Internet. The GRID-GUM2 runtime system [331] is an extension of GUM, and supports Glasgow Parallel Haskell. Considerable effort was required to modify the load distribution mechanisms compared with those for clusters, but the results have been promising. The authors demonstrated that provided the load balancing mechanism explicitly takes into account the latency between machines and any variations in processor speeds, it is possible to achieve good performance in these settings. Their work demonstrates that it is possible to implement parallel graph reduction efficiently in widely distributed environments [333, 332].

2.4.5.2 Frame organisation

The way in which concurrently evaluated parts of a program are represented can differ between implementations. Figure 2.9 illustrates the two main approaches, where each box represents a slot within a frame (function activation record) that contains either an unboxed value or graph pointer.

Implementations which use the *stack model* evaluate the program using a stack, in a very similar manner to that of imperative languages. A function call causes a new frame to be pushed on to the stack, and a function return causes the top frame to be popped from the stack. Parallel execution is achieved using multiple threads, each of which has its own stack, and evaluates expressions present in the graph. Implementations which use this model include GUM [300] and GRIP [162].

The other main approach is the *packet model*, in which each frame is stored as a separately allocated heap object, and contains enough space to hold temporary working variables for evaluating the function. References between different frames are maintained using pointers, just like references between nodes in the graph. This representation is sometimes referred to a a *cactus stack*, since it may have multiple branches instead of a single chain.

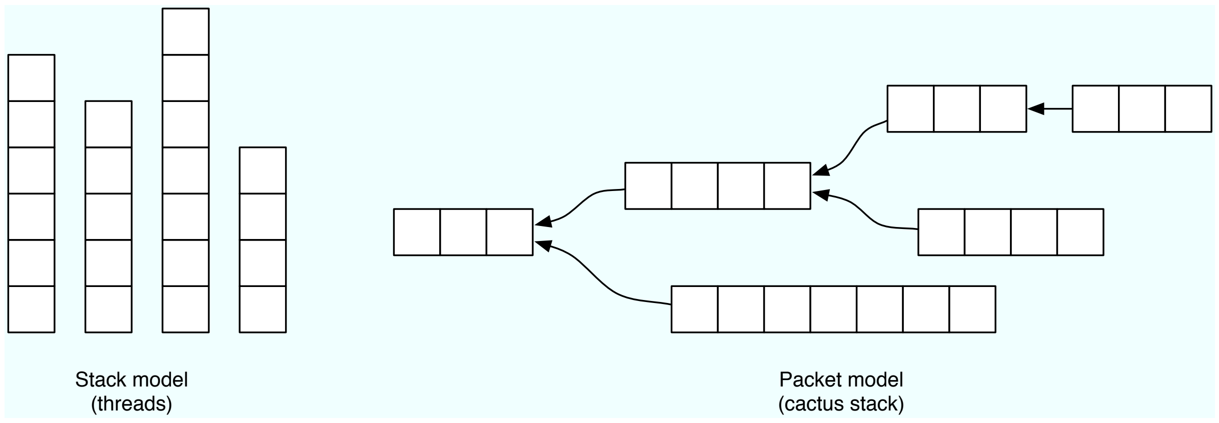


Figure 2.9: Frame organisation models

Parallel evaluation is achieved by having multiple frames active at the same time, with each available processor executing a separate frame. The order in which the frames are executed is constrained by their data dependencies, but as long as these constraints are satisfied, the results will be the same regardless of which frames complete first, due to the side effect free nature of pure functional languages. Implementations which use the packet model include ALICE [79], the <v,G>-machine [24], and the HDG-Machine [186].

One disadvantage of the stack model is that any time an expression needs to block, the thread cannot continue until the expression is unblocked. This can occur in a distributed memory implementation when a request is sent to a remote node to retrieve a value that is not in local memory [300], or when an RPC request is sent to a service (a facility supported in our own implementation). Each time this happens, if the processor does not have another thread waiting to execute, it must create a new thread to evaluate any sparks that are in the spark pool. If blocking occurs often enough, this would result in many threads being created. In contrast, the packet model deals with this more cleanly, since the processor can just switch to the another frame that is ready for execution, without the overheads of creating a new thread. The relative performance of each approach thus depends on both the regularity of blocking and the cost of thread creation. It is not clear from the literature which of these approaches is better in general.

The packet model has some performance costs due to reduced locality, but is simpler to implement. This simplicity comes from the fact that closures, which represent unevaluated expressions, use the same representation as frames. Converting a closure into a frame can thus be done simply by changing its type field. Some work has been done to optimise the representation of cactus stacks in order to improve locality by storing frames in a contiguous area of memory wherever possible [140, 122].

2.4.6 Input/Output

A core tenet of functional programming languages is the notion of *purity*, in which expressions and functions can only compute a result, rather than modify state. This presents a problem for I/O operations, most of which involve stateful interaction with the outside world. The lack of a guaranteed order of evaluation means that any side effecting actions taken within a pure expression would not necessarily occur in the order in which they are expressed. For example, two different output statements could be executed in either order, resulting in non-deterministic behaviour.

The approaches that functional languages take to supporting I/O are based on the philosophy that since I/O involves side effects, it can only be done outside of the "pure" world of expression evaluation [144]. Several different techniques exist for expressing this interaction, all of which require reads or writes to occur in a sequential, well-defined order. This effectively forces an imperative programming style on the portions of programs that need to interact with entities external to the program, such as the network or filesystem.

The three main approaches to I/O support are as follows:

- **Lazy streams** [241]. In this approach, the top-level function of a program is given a type of the form [Response] → [Request], i.e. a mapping from *responses* (received from the operating system) to *requests* (sent to the operating system). Lazy evaluation permits the response list to be passed to the program without any of its contents initially being evaluated, and the program traverses the list as it needs to process more responses. The program produces a list of requests, each of which asks for some action to be performed, such as reading from or writing to a file. For each request that is sent out, the language runtime executes the operation, and places another element in the response list, which the program can then inspect to get the result of the operation.

- **Continuations** [147]. This style utilises *continuations* (functions representing a suspended execution state), which are passed as parameters to I/O operations, and are invoked when the operation completes. The parameter passed to such a function is the result of the operation, so when the function is called, it can do whatever it needs to with this result. The following example uses continuations to read two characters from standard input, and then return them in a list:

  ```
  (readchar (λc1.readchar (λc2.[c1,c2])))
  ```

  In this example, the parameter to the first `readchar` call is a lambda abstraction (function) which will be invoked when the first character is read, with the value being passed in as `c1`. The parameter to the second `readchar` call is another lambda abstraction which will be called when the second character is read, whose value will be passed in as `c2`. This second lambda abstraction returns a list containing both characters, which will be the result of the expression as a whole.

- **Monadic I/O** [165]. This models all side effecting computations as *actions*, which take the "state of the world" as input, and return a modified state of the world plus a result value. Because this state cannot be meaningfully duplicated, only one instance of it can exist at a time, and thus multiple I/O actions must be sequenced together by passing the state returned by one as input to the next. Of course the entire state of the world is not actually stored in memory, but this is a useful way of thinking about side effects within the context of a pure functional language.

In all of the above approaches, functional programs that perform I/O must essentially be divided into two parts: an outer imperative shell which interacts with the outside world, and an inner core that consists entirely of side effect free expressions. Any data read from or written to files, network connections, or user interfaces must go through the outer shell and be passed to/from the inner parts of the program.

This problem was partially solved by Clean's I/O system [6], which models operations as transformations to only *part* of the state of the world. Each part represents some entity that is independent of others, such as an individual file. Sections of a program which access independent entities with separate state can run concurrently, as long as all operations on a particular state representation happen in sequence, which is enforced by *uniqueness typing* [274]. However, Clean still requires a sequential ordering of all top-level I/O operations such as opening files, which require a "world" object to be passed around.

An unfortunate consequence of the restrictive view of all I/O operations as impure is that those which *are* actually pure cannot be used within normal expressions. The fact that an operation involves interaction with another computer does not necessarily mean that it has to be ordered sequentially with all other such interactions. Those that are known to be side effect free could be quite safely used within the context of the pure world of expression evaluation. For example, a function that retrieves a web page and extracts part of its content does not involve side effects. Such interaction, however, is not allowed by any of the above models.

These sequential models of I/O are in fundamental conflict with parallelism. Data processing programs that need to perform read-only data retrieval over the network or invoke remote side effect free operations such as database queries or web page retrieval cannot utilise parallelism in pure functional languages that strictly adhere to this model. Although such network interactions do technically involve stateful operations such as connection establishment and read and write calls, these are simply internal implementation details of making a request; the actual operation as a whole can be considered free of side effects, provided that its execution does not involve modifications to any externally visible state. From the perspective of the programming model, read-only operations invoked over a network should arguably be permitted. Without this, the opportunity for combining one of the greatest strengths of functional programming — ease of parallel programming — with remote procedure calls is lost.

One of the main challenges with this is that it is usually not possible for a compiler to determine whether or not a particular remote operation can cause side effects or not. Were I/O operations permitted within pure expressions, the responsibility would fall on the programmer to ensure that those operations they invoke are indeed safe. This compromises the static safety guarantees that functional languages work so hard to provide. However, in some cases it *is* possible to infer from the semantics of the network protocol whether an operation can safely be considered free of side effects. For example, the HTTP specification [102] states that GET requests are supposed to be read-only; provided web servers comply with this requirement, such requests are safe to execute in any order.

Some functional languages do actually provide ways to perform "impure" operations within the context of a pure expression, but consider such features to be special cases that should not normally used. Haskell provides a function called `unsafePerformIO` which enables monadic I/O operations to be performed inside a pure expression, though its designers describe it as a "dangerous weapon" and warn against using it extensively [167]. However, data processing programs that involve parallel invocation of remote operations fundamentally rely on such mechanisms, so it can be argued that for these types of programs, such operations should be considered safe and normal.

If we are to use functional programming as a model for workflows, we need to be able to make service calls within the context of pure expressions. Use of workarounds like `unsafePerformIO` is undesirable when almost all of the operations contained in the workflow correspond to remote services. In Section 3.5.8, we describe how our own programming model exposes network connections as side effect free functions which transform from input to output. This enables network communication to occur within the context of a pure expression, and service calls to be treated as candidates for parallel evaluation.

2.4.7 Relationships between functional programming and workflows

Data-oriented workflow models like those of Taverna and Kepler have much in common with functional languages. The idea of taking a series of inputs and invoking operations on them based on data dependencies is the same basic philosophy used in writing functional programs. Importantly, this lack of side effects makes it possible for the language implementation to automatically determine which operations can be safely executed in parallel. However, workflow languages typically lack several important features that are found in functional languages, such as higher-order functions, recursion, list processing, and support for fine-grained computation.

Workflows are specified using a graphical notation, which is edited visually or by writing a series of statements identifying individual nodes and edges in the graph. In contrast, functional programs are constructed out of expressions, which are much more concise, since they permit several operations to be specified per line, and the edges are implicit in the syntactic structure. Section 3.6.4 compares these two representations in more detail.

Both types of languages use graphs during execution. Workflow languages are based on variants of the dataflow model, where nodes represent operations, and edges indicate data dependencies. Functional languages often use graph reduction, where nodes represent values, lambda abstractions, and function applications. Graph reduction treats functions as a particular type of value, enabling them to be used with higher-order functions, which greatly enhance the expressiveness of a language by enabling certain classes of control structures to be defined in user code. Dataflow languages require these higher-order functions to be supported natively as part of the programming language, complicating the implementation and limiting the flexibility of the language.

Most workflow languages support the invocation of remote operations provided by other machines on a network. This implies that it is necessary for the workflow engine to perform I/O operations to exchange data with these remote tasks. Functional languages do not permit I/O within pure expressions, based on the assumption that all I/O involves side effects and must therefore be forbidden in situations where the order of evaluation is not explicitly defined. This prevents parallelism from being exploited by workflows implemented in existing functional languages.

Several cases of using selected ideas from functional programming in the design of workflow systems can be found in the literature. One such approach is the use of relational databases with active views representing a workflow [271, 203]. This enables results of workflows to be queried with SQL, which is similar in some respects to functional languages due to its declarative nature and the lack of side effects in SELECT statements. However, this work requires the workflow structure to be specified in terms of a data definition language, which is substantially different to functional languages.

Martlet [123] contains language constructs similar to the `map` and `foldr` higher-order functions found in functional languages, but is based on an imperative programming model in which the expansion of loop bodies happens at compile time. It only permits one such level of expansion, limiting the range of algorithms that can be expressed. MapReduce [83] integrates the concept of `map` and `reduce` functions into the design of the system, leveraging the parallelism that these make possible. However, it enforces a fixed structuring of these primitives, with no scope for other higher-order functions to be used.

Gregory and Paschali [126] describe an extension to Prolog [196] which supports the definition of workflows. This extension involves an `rpc` rule, which takes three parameters: destination, message, and result. The destination and message parameters must be bound to values at the time of the call, while the result parameter must be unbound, as it will later be bound to the result of the operation upon completion. The `rpc` rule operates asynchronously; once it is called, execution of subsequent statements continues. Multiple service calls may be in progress at the same time. Blocking occurs when an attempt is made to access an as-yet-unbound result variable; the calling thread is woken up once the result arrives from the service.

This asynchronous invocation technique is similar to the way in which we support parallel invocation of multiple services. However, our work is based on a functional programming model, while Prolog is based on a logic programming model. We argue that functional programming is more appropriate for workflows, since it focuses on *performing computation* based on the composition of functions, whereas logic programming focuses on *searching* a collection of information for values matching particular criteria. Additionally, functional programming has a much closer relationship to the task dependency models of existing workflow languages.

Two of the workflow systems described in Section 2.3.7 have been the subject of comparisons with functional languages. In contrast with the informal descriptions usually given for workflow languages, Turi et. al. [306] defined the semantics of Taverna's programming model in terms of *computational lambda calculus* [232]. The advantage of this approach is that it provides a formal, unambiguous definition of the language, and also opens up the possibility of automatic translation from Taverna's programming model into the target language. One drawback of the work is that it uses operational semantics; we believe that denotational semantics would have been a better fit, as it would have emphasised the meaning of each language construct rather than the way in which it is executed. Another drawback is that the authors assume the existence of certain primitive operations such as dot products and list flattening which are not present in pure lambda calculus, ignoring the fact that these can in fact be expressed in terms of lambda calculus itself using techniques such as cons pairs [221, 110] to represent lists. Nonetheless, it indicates interest in using functional languages to express the semantics of workflows.

Ludascher and Altintas [209] reported on an experience with taking a workflow originally created with Kepler, and re-implementing it in Haskell. They demonstrated the benefits of Haskell's programming model and syntax, which enabled the workflow to be implemented in a more natural and concise way than it could be in Kepler. In particular, higher-order functions were shown to be useful for providing abstraction mechanisms that allow low-level control flow logic to be separated from the basic structure of the workflow. The conclusions of this paper were that workflow systems could benefit from adopting features found in functional languages, such as declarative iteration constructs like `map` and `zipWith`. The authors also mention the issue of data transformation, suggesting that this could usefully be done in embedded scripting languages such as Perl and Python, but do not discuss the possibility of using the data manipulation features already supported by Haskell for this purpose.

From the perspective of functional programming, a couple of attempts have been made to incorporate workflow functionality into functional languages. One approach is HAIFA [108], a library for Haskell which adds support for XML and web services, and is designed for expressing state-oriented workflows. This suffers from the sequential I/O restrictions described in Section 2.4.6, and is thus unsuitable for data-oriented workflows.

Another approach is *SymGrid-Par* [132, 330, 334], which uses parallel functional programming techniques to coordinate computations carried out by *symbolic computation* systems such as Maple [112] and Mathematica [324]. It is based on the GRID-GUM2 runtime system [331] mentioned in Section 2.4.5.1, which uses parallel graph reduction to execute programs written in Glasgow Parallel Haskell [302] using a collection of machines. Each individual machine hosts an instance of the runtime system as well as the symbolic computation software; interaction between the two is achieved using an RPC-like protocol. All communication between machines occurs using the graph distribution mechanisms of GUM, much like our choreography mode discussed in Section 3.4.2.

SymGrid-Par can be considered a workflow system, because it provides a similar form of task coordination to the workflow systems described in Section 2.3. However, its authors neither use the term *workflow* to describe it, nor provide any significant comparison with other workflow systems. We take this as further evidence of the lack of interaction between the workflow and functional programming communities, despite the great deal that the two have in common. Although SymGrid-Par only emerged after our own implementation was mostly complete, and was created to serve different goals to our own, we take its existence as further validation of our argument that functional programming languages are ideal for expressing workflows.

Our work extends the ideas of SymGrid-Par in several ways. Firstly, we support access to remote services which reside on separate machines to the workflow engine itself. Secondly, our programming model supports automatic parallelisation, and enables developers to access services using any protocol. Thirdly, we explore the use of a XQuery for expressing workflows involving web services and XML processing. Finally, our study includes a comparison of the functional programming model with existing workflow models, as well as discussions of performance issues specific to workflows.

2.4.8 Discussion

Functional programming languages lend themselves to parallel execution because of their side effect free nature, which makes it possible to safely reduce multiple independent expressions in parallel. A large amount of work has explored this idea, with numerous implementations produced for both shared memory and distributed memory systems. In the early days of research into parallel functional programming, it was hoped that purely automatic techniques could be used to parallelise code without any explicit direction from the programmer. However, much of the work done in this area since the early 1990s has recognised the limitations of this approach, instead relying on programmer-supplied annotations.

One of the key challenges with automatic parallelisation arises due to granularity. Virtually all of the research on parallel functional programming to date has focused solely on fine-grained parallelism, with the implicit assumption that all of the application's code will be written in a functional language. In attempting to extract the maximum possible performance from the hardware, language designers have avoided usability features which harm performance too much. In particular, automatic parallelisation of fine-grained operations is considered impractical, due to the runtime overheads of managing the excessive number of small tasks that it tends to generate.

For coarse-grained operations with restricted control structures, automatic detection of parallelism has already been demonstrated by many of the projects described in Sections 2.1, 2.2, and 2.3 to be perfectly viable. Some task farming systems, such as SETI@home [15], are capable of scaling to tens or hundreds of thousands of machines. For the trivial case of independent tasks this is obvious, and many graph-based workflow systems which operate on a similar model to functional languages are also able to achieve parallelism without explicit annotations. Recent work by Google has demonstrated the use of the well-known `map` and `reduce` constructs to index the entire web efficiently and in parallel [83]. These efforts demonstrate that automatic parallelisation is achievable, *provided that most of the computation is carried out by coarse-grained operations.*

Although functional languages have many attractive properties, they retain only niche status in the software development community. The vast majority of code is written in imperative languages, and programmers looking for ways to parallelise this are much more likely to adopt technologies which allow them to re-use their existing code, instead of requiring them to translate it into a different language. This is one of the reasons behind the popularity of the various distributed computing systems described previously, including workflow languages. With these systems, programmers expose their existing imperative code as tasks, and provide a high-level specification of the tasks to be invoked in parallel. We believe that if parallel functional languages were to support this mode of development, they would be much more useful for developers with lots of existing code written in imperative languages.

The emergence of distributed computing and workflow technology as a way of exploiting parallelism across multiple machines opens up new opportunities for the application of functional programming theory. Rather than attempting to efficiently parallelise fine-grained computation, our work focuses on the use of functional programming languages for expressing workflows involving coarse-grained operations exposed as services. In this context, we sought to determine if it is possible to implement automatic parallelisation without impacting the performance-sensitive parts of the application, which are executed externally. Importantly, this does not rule out the implementation of *some* computation within the workflow, provided it represents only a very small portion of an application's overall computation. As discussed in Section 2.3.8, it is often useful to include some amount of application logic and data manipulation directly within a workflow.

Our approach enables the strengths of both functional and imperative languages to be combined. High-level aspects of an application, which are not performance sensitive, may be expressed in a workflow language based on a functional programming model. This enables the runtime system to take advantage of implicit parallelism without requiring the programmer to use error-prone synchronisation constructs. Performance sensitive, fine-grained computation may instead be implemented in compiled, sequential languages such as C and Java, and exposed as services which are invoked by the workflow. The use of functional programming as a model for workflow languages promises to increase the expressiveness with which distributed computations can be constructed, by leveraging existing work from the functional programming community in a new way.

2.5 Web services

The term *web services* [305] refers to a set of specifications which may be used to construct distributed applications. These specifications build on common Internet technologies such as HTTP and XML to provide high-level, platform-independent mechanisms for two or more pieces of software to interact over a network. Web services are widely used for many purposes, such as business-to-business integration and providing access to specialised software over the Internet. They are the latest in a long line of client-server technologies that includes SUN-RPC [284], DCOM [270], RMI [285], CORBA [240], and others [292].

Unlike with many previous efforts, platform independence is a key goal of web services. A client written in a particular programming language and running on a particular operating system may interact with a server built using a different programming language and platform. The fact that both sides speak the same protocol means that neither needs to know or care how the other is implemented. While this is true of all widely used Internet

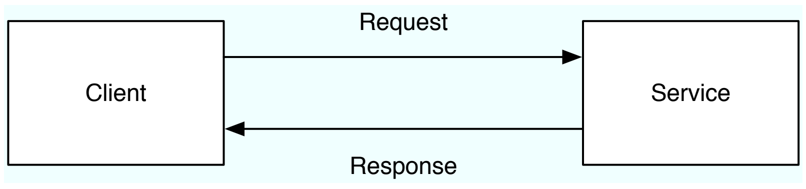


Figure 2.10: Client/service interaction

protocols, web services make interoperability easier by enabling toolkits to abstract away low-level communication details, and provide high-level facilities for use by applications.

2.5.1 Concepts

Web services follow the same basic remote procedure call model that has seen many incarnations over the years. The main distinguishing feature of web services are the specific protocols and data formats used, which are designed to aid interoperability and integrate well with existing web infrastructure. Before going into detail about these standards in Section 2.5.3, we will first outline the basic concepts behind web services and other similar systems.

A *service* is a piece of software hosted on a server, which is executed upon request from a *client*. The client sends a *request message* containing the identifier of the service, the name of the operation, and any input parameters. The service executes the requested operation, and then sends a *response message* back to the client containing the result. This interaction is shown in Figure 2.10.

The set of operations provided by a service is defined in an *interface*. For each operation, this specifies the name, input parameters, and result. Each input parameter and result has a specific data type associated with it, which may be a simple atomic type such as integer or string, or a more complex data structure, also defined in the interface. The role played by an interface is as a contract between the client and service, as well as a formal definition of the messages accepted by the service that toolkits can use to generate support code. Interfaces may also contain protocol-specific information specifying how to invoke operations.

When processing data, the client and server may either deal with the data directly, or have it converted into language-specific data structures. In the former case, the request and response messages may be accessed as binary or plain text, or as a parse tree based on a defined message format. In the latter case, support code provided by the runtime environment *marshals* data to or from the language's native data representation. This marshaling process may involve information loss if the message format and language type system do not match.

On the client side, operations may be invoked using a *remote procedure call* mechanism, in which the client application calls a function which performs the necessary marshaling and network interaction with the server. The function's code is generated automatically by a support tool based on information obtained from the service's interface. This abstraction enables remote functions to be treated as local ones, although latency and the potential for network failures must still be taken into account by the programmer.

2.5.2 Service oriented architecture

Web services are often used in the context of *service oriented architecture* (SOA) [223], a systems design approach in which applications are built out of multiple services located at different sites. This is essentially the distributed equivalent of component-based software development, where each service corresponds to a component, and applications utilise multiple components. Workflows, discussed in Section 2.3, are useful in the context of SOA as a way of building applications that utilise functionality provided by multiple services.

SOA is increasingly being chosen by many as an alternative to tightly-coupled, proprietary systems. The platform independence of web services reduces the dependence between clients and servers, which may utilise different programming languages or operating systems. Many businesses and scientific researchers are choosing this approach for constructing loosely-coupled systems which enable different technologies to be integrated with relative ease.

2.5.3 Standards

Two styles of web services are in common use today. SOAP (Simple Object Access Protocol) [277] services primarily use the web as a transport mechanism, and define application-level semantics on top of the HTTP protocol. They have the advantage of providing interface definitions that indicate which operations are available, and the types of parameters accepted by those operations. REST (REpresentational State Transfer) [258] services are more closely aligned with the architectural principles of the web, in particular the use of URLs to refer to objects, and HTTP methods to indicate actions to be performed. This approach has the advantage of being simpler, although it lacks mechanisms for defining the interface provided by a service. REST services are discussed further in Section 2.5.7.

Here, we focus primarily on SOAP services, which make use of the following standards:

- **XML (eXtensible Markup Language)** [50] provides a structured format for representing messages, as well as parameters and return values exchanged between a client and service.
- **HTTP (HyperText Transfer Protocol)** [102] is used as the transport protocol, enabling web services to be invoked in a similar manner to dynamic web pages. Host environments for web services act as a web server, and may therefore serve up other content as well. The use of HTTP also enables service endpoints to be identified by URLs.

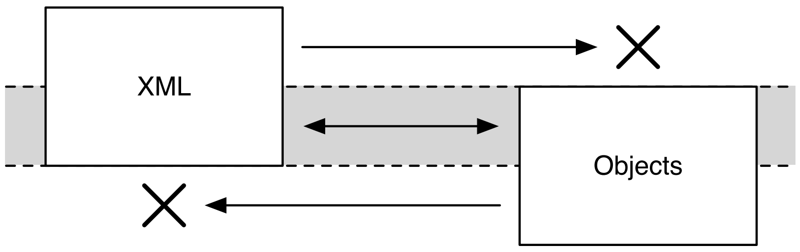


Figure 2.11: The X/O mismatch

- **SOAP (Simple Object Access Protocol)** [47] defines the structure of request/response messages exchanged with a service. It specifies how the operation name and parameters are to be encoded, as well as an envelope structure that contains the message body as well headers containing metadata.
- **WSDL (Web Services Description Language)** [66] is used for defining service interfaces. Like SOAP, WSDL is based on XML, and specifies elements used to define operations, input/output messages, and protocol information necessary for invoking operations.
- **XML Schema** [98] is a type system for XML which, in the context of web services, is used for defining the types of parameters and results for service operations. WSDL files include references to XML Schema definitions for any complex data structures that a service uses.

Many more standards have been defined for higher-level functionality relating to web services. However, all but those above are considered optional; the ones we have listed here are sufficient for implementing the common case of a client invoking operations on a service.

The main benefit of these standards is that they provide a widely accepted syntax and protocol structure for which different web service toolkits and hosting environments can be implemented. This support software largely removes the responsibility of dealing with low-level details from the programmer, although it is still useful for a developer to have a reasonable understanding of how they all work. WSDL and XML Schema, in particular, are useful for enabling toolkits to generate the client-side stubs and server-side skeletons which implement the remote function calls and marshaling.

2.5.4 Language support

A large range of bindings are available for different languages to enable them to interact with web services. On the client side, the main responsibility of these bindings is to expose the service as a *stub object* upon which methods can be called to invoke operations. On the server side, the bindings provide integration with a web server to handle requests from clients and invoke the appropriate functions. Marshalling is performed by the bindings on both ends.

Toolkits which take this approach include Apache Axis [251] and JAX-WS [189] for Java, gSOAP [95] for C/C++, SOAP::Lite [191] for Perl, ZSI and SOAPpy [173] for Python, SOAP4R [266] for Ruby, and Windows Communication Foundation [224] for .NET.

One of the problems with marshaling is that very few programming languages have a type system that is completely compatible with XML, and thus the marshaling process cannot handle all data. For example, not everything that can be represented in an object-oriented language can be expressed as XML, and vice-versa. This is known as the *X/O mismatch*, depicted in Figure 2.11, and is discussed at length in [228] and [192].

Examples of XML features which do not translate well into objects are multiple child elements with the same name, mixed content containing both elements and text, and the distinction between child elements and attributes. In the other direction, cyclic graphs and stateful objects like threads and file handles cannot be represented in XML. An implication of the X/O mismatch is that in practice, web service developers must restrict themselves to using only the undocumented subset of XML that is supported by all the major languages and toolkits.

A solution to this problem is to have the clients and services deal directly with the XML content, bypassing the marshaling process. The most common way to do this is to access the XML parse tree using the DOM (Document Object Model) API [142]. By using the API to traverse the tree structure of the SOAP message and extract the required fields, there is no possibility of information loss, since the full XML content can be accessed in this manner. However, the DOM API is considerably more awkward to use than built-in language facilities, making this an unattractive option.

As an alternative to DOM, several recent efforts have explored the idea of extending existing languages with first-class XML support, enabling XML documents to be constructed and accessed using native language constructs. Examples of such extensions include Cω [39], XJ [134], XTATIC [111], and LINQ/XLinq [227, 226], the latter of which is being incorporated into new versions of C# and Visual Basic. These solve the problem of making XML easily accessible, and are useful for a wide range of XML processing tasks. However since these techniques are quite recent, their incorporation into web service toolkits has yet to be explored.

Another alternative is to write the client and/or service in a language whose data model is completely based on XML. Examples of such languages include XSLT [179] and XQuery [43], both of which are supersets of XPath [36]. As with the language extension approach, this would also allow direct access to and construction of XML data directly within the language. Additionally, the fact that the language is solely based on XML removes the possibility of some data structures not being serialisable to XML.

However, neither XSLT or XQuery currently include support for web services. Some work has been done on using the latter for web service access, which we discuss in Section 2.6.1, but this functionality is neither part of the standard nor widely implemented. Both are also side effect free functional languages, making them unsuitable for applications involving state manipulation. The functional nature of these languages, however, opens up the possibility of automatic parallelisation, an issue we explore along with web service access in our own XQuery implementation, described in Chapter 5.

2.5.5 Asynchronous RPC

The most common programming technique for clients of web services is synchronous remote procedure call. In this model, each remote operation is exposed on the client side as a function, which is invoked using the same syntax as a local function call. This has the advantage of simplicity, because the programmer can treat remote operations just like they were local ones. All of the details relating to interaction with the server and marshaling parameters and return values are taken care of by the automatically generated stub objects.

A problem with this approach is that remote calls cause the client to block [290]. If a remote operation takes a long time to complete, the client will sit there idle, unable to do anything else. For interactive programs, this makes the user interface unresponsive, and for processing-intensive client programs, it can waste otherwise useful CPU time. Ideally, the client should be able to continue on with other things while it is waiting for a remote operation to complete.

One solution to this problem is to make each remote call in a separate thread. This way, local processing can continue in the main thread of the program, and the blocking of other threads will not affect the local computation. A limitation of this approach is that in most runtime environments, threads are expensive, so it is not practical to have a large number of outstanding calls to different servers in progress at the same time. It can also complicate the client logic, because the programmer must explicitly deal with thread creation and synchronisation.

Another approach is to use asynchronous remote procedure calls [13]. In this model, the client program initiates a request, and then continues on with other processing immediately. The runtime system takes care of receiving the response from the server, and then notifies the client program when it becomes available. This may be done with a *future* object [97], on which the client can invoke a blocking *get* operation when it needs the result of the service call. Alternatively, the runtime system may support the use of programmer-supplied callback functions [315] that are invoked upon receipt of the response, so that the client can choose what do to with the result as soon as it arrives. An example realisation of the callback approach is the support for sending asynchronous HTTP requests provided by modern web browsers, which is used in the AJAX (Asynchronous JavaScript And XML) programming style for web applications [246].

These asynchronous remote procedure call and multi-threading techniques both share the disadvantage that they complicate the programming logic on the client side. It would be more convenient for programmers if they were able to use the normal function call mechanism, but without blocking the whole program. However, imperative languages use a stack-based representation of execution state in which only one function call for a given thread can be active at a time. The stack approach means that if the active frame makes a remote call, then the whole thread is blocked, and frames lower down in the stack cannot continue with their execution until the one at the top completes.

A more flexible approach is to use a graph instead of a stack to represent function activation records (frames). Instead of organising them as a single list of frames, each time a function call is made, it is possible for the caller to continue executing, possibly invoking other function calls. This essentially treats each function call as a thread, though by implementing these at user level within the runtime system it is possible to avoid the costs associated with operating system-level threads. The graph-based representation allows multiple function calls to conceptually be active at the same time, so if one blocks on a remote call, then it is possible for others to continue.

In fact, this model corresponds exactly to the cactus stack model described in Section 2.4.5.2, which is often used for parallel functional language implementations. It is really only suitable for side effect free languages, because having multiple function calls active concurrently would introduce many synchronisation issues for imperative languages. Without side effects, multiple frames can safely be added to the graph and make separate blocking remote procedure calls at the same time.

Direct implementation of this approach requires a different execution mechanism to that used by imperative languages, and thus it is necessary to take this execution model into account in design of a virtual machine. This is one of the reasons why we have developed our own virtual machine in this project, rather than utilising an existing one such as the JVM (Java Virtual Machine) [202] or .NET CLR (Common Language Runtime) [48]. By having the level of control over execution that is made possible by a custom virtual machine design, we are able to exploit the graph-based model to efficiently support multiple concurrent RPC calls while retaining the usual function call semantics that other toolkits provide. The main difference with traditional RPC systems is that our approach requires the language to be side effect free.

2.5.6 State

An important issue in the design of web services is whether and how to deal with state. A web service may be either *stateful* or *stateless*, depending on how it is implemented. A stateful web service maintains information that persists across invocations, and may be modified by each. A stateless web service has no such information, and each invocation is a completely unrelated interaction to the others. The latter is typically the case for read-only interactions with a web server, such as retrieving a web page.

The issue of state in the context of the web can sometimes be confusing, because although HTTP provides no explicit notion of state, it is possible to build higher-level mechanisms which do enable stateful interactions. In the context of interactive web sites, cookies are routinely used as a *session handle* that correlates different requests from the same client, enabling users to conduct a long-running interaction with a site. Similarly, web services can achieve the same effect by having a handle to a session or piece of state passed in request/response messages.

When discussing web services in the context of data-oriented workflows, it is important to take into consideration the fundamentally side effect free programming model in which these workflows are expressed. Because a data-oriented workflow orders operation invocation based purely on data dependencies, there is no way to handle hidden dependencies between operations that rely on one method invocation being able to access state that was modified by a previous invocation. For this reason, web services used within these types of workflows should not maintain state, and only perform computation based on the input data provided with each request.

2.5.7 REST

An alternative style of web services to the SOAP model is REST (REpresentational State Transfer) [103, 256], which emphasises the use of existing web protocols and conventions over newer standards. The basic philosophy behind REST is the use of URLs to reference objects, and HTTP operations to perform particular actions on those objects. This is in contrast to SOAP's model, which identifies services by WSDL-defined endpoints, and does not directly support the notion of object references.

One of the core principles of web architecture is the use of a URLs to refer to the location of objects on the Internet. Usually these objects are web pages or media files like images, however they can in fact be any type of entity that can be made accessible by a web server. To invoke a method on an object, a client sends a HTTP request for the URL, with the method indicating what action they wish to perform on that object. For a web page, this method is GET, which means that the client wants to retrieve the contents of the object. GET can be applied to any object which has an electronic representation; the web service simply makes the contents of the object available to clients via this request mechanism.

HTTP provides other methods that can be used to act upon an object. The PUT method indicates that the object's content should be replaced; the client includes the new content in the request, and then this becomes the data stored on the server relating to that object. Similarly, DELETE removes the object. POST is a more general method which can be used to perform other types of actions on an object, such as updating part of its content or interacting with it in some other way. This is commonly used for web forms, where a browser needs to invoke some piece of server-side functionality with a series of input values.

Building distributed systems using REST involves modeling an application using the same concepts that are applied to web pages — that is, the use of URLs and HTTP to reference and act upon objects. This is a somewhat different design philosophy to SOAP services, which are closer in nature to traditional remote procedure call architectures. Thus it is not simply a choice of protocol or syntax to be made, but rather a design approach that is to be used when building an application. Because of its object/method focus, REST is more closely aligned with object-oriented software development techniques than SOAP services. With the latter, there is no inherent notion of an object, so objects can only be supported as an application-level concept. The closest one can get to invoking a method on an object with SOAP is by passing a value identifying the object as an argument to a service call.

A major advantage of REST is simplicity. SOAP and related standards are regarded by many to be unnecessarily complex, at least for the sort of applications that the majority of developers need to construct. This complexity, combined with vendor politics and the constantly evolving nature of standardisation efforts, has led some developers to chose simple, well-understood solutions which are less sophisticated but are nonetheless sufficient for many purposes. Proponents of REST [125] argue that it is better to utilise existing, proven standards and build upon the existing interaction models of the web, than to use HTTP as a transport protocol for alternative interaction models that resemble more traditional RPC systems such as CORBA.

Despite REST's advantages, it has the significant drawback that it does not provide any way to define the interface for a service. With the SOAP model, a service provider makes an interface definition available in the form of a WSDL document which lists the operations provided by a service, and the format of the request and response messages it is able to process. This can be used by development tools to generate client stubs and server skeletons to marshal data between local objects and the on-the-wire format. Additionally, the service definitions provide a formal description of the contract between a service and its clients, something that is not provided by the REST model. Although there are significant complexities associated with SOAP and WSDL, these can largely be dealt with by supporting software, and in some cases the benefits of formal interface definitions and error handling mechanisms can make this model more attractive.

2.5.8 Applicability to workflows

Web services are particularly useful in the context of workflows, where functionality provided by different machines within a network is coupled together to achieve higher-level functionality. The goals of service oriented architecture are to enable applications to be constructed in such a manner, using standardised protocols to increase the ease with which systems implemented in different programming languages and running on different platforms can be integrated. Workflows represent one particular case of such applications.

Several of the workflow systems described in Section 2.3.7 support the ability to invoke web services. Multiple services on different machines may be used together by connecting the outputs of some services to the inputs of others, such that the flow of data between them is dictated by the workflow structure. From the perspective of the programming model, each service operation is equivalent to a function invoked by the program. The data-oriented functional programming model enables multiple function calls to be executed in parallel, which in the case of web services corresponds to multiple asynchronous requests being in progress at the same time.

Perhaps the biggest advantage of web services is their widespread support and recognition within the industry. Because of the availability of many development tools and middleware systems, an increasing number of applications are being constructed in such a way that they are accessible as web services. The standardisation efforts that have been made in this area enable many different applications to be used together by allowing the low-level syntactic and protocol details to be abstracted away by support software.

Of course, integration of different services will always require manual programming effort, because the semantics of each service and the ways in which they can be used together are inherently application-specific — this aspect cannot be addressed by standardisation efforts. It is thus very important for service composition languages to provide developers with sufficiently flexible language capabilities, including the ability to express arbitrary computation and data manipulation logic involving data exchanged with services. As discussed in Section 2.3.8, many workflow languages are sorely lacking in this respect. In order to meet these goals, we look to XQuery as a solution, due to its native support for XML, expressive language constructs, existing widespread usage, and the fact that it is based on a functional programming model.

2.6 XQuery

Web services use XML as a data encoding format. In order to construct workflows that compose together web services and perform intermediate computation on the data values exchanged between them, it is therefore necessary for the language to support XML. In most programming languages, this support consists of libraries that expose APIs providing access to an XML document as a tree; the most common example of this is DOM API, mentioned in Section 2.5.4. These APIs are awkward to use, due to the explicit and manual nature of many operations such as tree traversal. As an alternative, several high-level languages have been developed which provide native language support for XML.

One of the best known examples of such a language is *XQuery* [43], a pure functional language which has been standardised by the World Wide Web Consortium. The language provides a number of useful constructs for accessing parts of an XML document, as well as generic control flow and expression constructs similar to those found in other programming languages. The XQuery language specification does not include support for web services, but several researchers have recognised its potential as a workflow language [90], and conducted initial studies into how it could be extended with this support. We discuss this work further in Section 2.6.1.

Although the idea of using XQuery to access web services is not new, we explore it in our project as a demonstration of how a suitably generic programming model, which we describe in Chapter 3, can be used as a target for compilers of high-level workflow languages. Previous work has addressed service access and in some cases parallel evaluation at the level of the XQuery language itself; our approach is to instead have these aspects implemented at a lower level by a virtual machine, and to implement XQuery by compiling it into an intermediate language which can be executed on the virtual machine. These details are discussed further in Chapter 5.

Implementations of XQuery include Saxon [177], Qexo [45], Galax [257], XQPull [99], Oracle XMLDB [205], SQL Server [244] and the BEA streaming XQuery processor [104]

2.6.1 Existing approaches to web services integration

There is a natural fit between XQuery and web services, due to the fact that they both use XML as a means of representing structured data. Web services accept and produce messages in XML format, and XQuery provides language features that make it easy to process XML content. We believe that the reason why current versions of XQuery lack support for web services is due to the designers' focus on supporting database queries rater than information integration over the Internet. However, several researchers have proposed techniques for integrating web services or similar technologies with the language.

One such proposal is XRPC [336], a language extension containing RPC mechanisms designed specifically for communicating with XQuery code on a remote machine. Although it uses SOAP as a transfer protocol, it is not technically based on web services in the true sense, as the messages are encoded in a proprietary format that can only be understood by recipients that are also written in XQuery and use the same language extension. Limited support for parallel requests is provided, but only within individual loops in which service accesses are made. Our approach is more general than this, permitting interaction with arbitrary web services, and supporting parallelism in a way that applies in a generic manner to all XQuery language constructs.

Dynamic XML documents [3] are proposed as a form of XML document which includes elements which indicate that a portion of the document can be obtained by calling a web service with a given set of parameter values. When the document is accessed during evaluation of an XQuery expression, and traversal reaches a web service element, a call is made to the corresponding service to obtain the required data. This data may then be further inspected or traversed by the query. This idea is similar to the one we pursue in this thesis, except that ours is a more general model which in which service calls are made from within XQuery expressions, rather than in the XML document that is supplied to the query. Our approach permits computed values to be passed as parameters, and for the results of service calls to be passed to other service calls. In contrast, dynamic XML documents require service parameters to be specified statically, and expose results as part of the input document which can be queried but not passed to other services.

An XQuery extension proposed by Onose and Simeon [243] is the `import service` statement [90], with which a programmer specifies the URL of a service they want to access, and associates a namespace prefix with this URL. The implementation makes the operations of this service available as XQuery function calls with the specified prefix. This mapping, which is implemented as an extension to the Galax XQuery processor [257], is based on an additional built-in function that enables SOAP requests to be submitted via HTTP. When a web service is imported, a stub module is generated containing a function for each remote operation. Each of these functions generates a SOAP envelope, submits it using the built-in function, and then extracts the result value from the SOAP response. This implementation also supports the inverse situation, where XQuery modules are exposed as services, such that each function in the module can be invoked by a client. We support the `import service` statement in our own implementation, as discussed in Section 5.3.1.

Another implementation with similar goals to our own work is WSQuery [129], which adds a `callws` function to XQuery which may be used to invoke a web service based on a supplied URL, operation name, and set of parameters. This implementation also supports parallel execution of queries, so that multiple machines can be used for executing service operations, and latency hiding can be achieved when multiple services are invoked.

The scheduling technique used for parallel execution is based on a dynamic unfolding of the graph, such that web service calls within `for` loops and `if` statements are scheduled at runtime once the number of calls becomes known. While this work addresses similar goals to ours, the implementation technique used is quite different, in that parallelism and scheduling are handled at the level of specific XQuery constructs, whereas we instead handle parallel execution at a lower level within the generated intermediate code.

Finally, XQuery itself includes a `doc` function [214] which may be used to retrieve the contents of a specified URL. This function can be used from within any expression, and, when called, returns an XML document which can then be accessed and traversed using the standard XQuery language constructs. This means that a query can make use of externally-available information provided by a web page. In a restricted sense, this can be used to access the subset of REST services which are accessible via HTTP GET, provided they accept parameters supplied as components of the URL, such as the query string or path components. However, SOAP-based web services and some REST services require that HTTP POST be used instead, which is not supported by the `doc` function.

2.7 Summary

This chapter has discussed several areas related to programming technologies and distributed computing that have complementary properties. While these have largely been treated as separate research areas to date, we have identified ways in which ideas from each of these areas may be combined in order to provide useful capabilities to developers of distributed applications.

Workflow systems are a form of distributed computing middleware which provide a high-level programming model for coordinating task execution. They are often based on a data-oriented model, in which tasks are free of side effects, and the structure of the workflow is expressed in terms of data dependencies between tasks, enabling automatic detection of parallelism through analysis of the graph. Built-in support for parallel execution and remote service access makes workflow systems an attractive choice for implementing large-scale parallel and distributed data processing applications. However, these benefits have come at the cost of language expressiveness, severely limiting the degree to which internal computation can be included directly within a workflow.

Functional programming languages are based on a similar data-oriented programming style to workflow languages. They support arbitrarily complex fine-grained computation, and are well-suited to parallel evaluation, due to their side effect free nature. However, existing functional languages provide neither automatic parallelisation nor access to remote services within the context of pure expressions, making them unsuitable for use as workflow languages. There is a great deal of existing theory and implementation experience in the functional programming community which we believe can be leveraged to develop more flexible workflow engines which support both coordination of external services and internal computation.

We believe that workflow technologies stand to benefit greatly from the influence of functional programming theory and techniques. Many issues that have yet to be adequately addressed by the workflow community have already been solved in the context of functional programming. Most significant among these is language expressiveness, which functional programming languages address through the use of lambda calculus as their underlying model of computation. Additionally, implementation techniques for efficient execution and concurrency can be applied to the construction of workflow engines, as we demonstrate in later chapters. In this thesis we attempt to provide the missing link between functional programming and data-oriented workflows by applying ideas from the former in the context of the latter.

An important issue for workflow languages is the network protocols and data formats supported. With the emergence of web services and XML as popular technologies for constructing service-oriented applications, a demand exists for workflow languages that cater for these standards. Although several existing workflow languages support web services, the programming models of these languages are very limited, and provide little or no support for manipulating XML data. XQuery is a highly expressive functional programming language designed specifically for processing XML data, and is thus well-suited to the task. Existing versions of the language do not include support for web services, though extensions have been proposed which add this support.

In the next chapter, we describe a programming model we have developed which incorporates ideas from functional programming theory to address the requirements of workflows.

Chapter 3

Workflow Model

This chapter presents a model for data-oriented workflow languages that supports not just *coordination* of external tasks, but also internal *computation* involving data produced by these tasks. Our approach extends the role of workflow languages to expressing parts of the application logic that would otherwise have to be implemented as external shim tasks written in other programming languages. Our model addresses this goal by providing an appropriate set of primitive operations for data manipulation, as well as a suitably general set of language constructs in which a wide range of algorithms can be expressed. This model is based on functional programming, and specifically the theory of lambda calculus, which we argue provides an ideal theoretical underpinning for workflow languages.

Although existing workflow systems are attractive due to their support for automatic parallelisation and integrated service access, they achieve these benefits at the cost of language expressiveness. Many common features which are simply taken for granted in most programming languages are missing from workflow languages, forcing developers to seek alternative ways of implementing parts of their workflow. Data structure manipulation, arithmetic evaluation, string manipulation, and data format conversion are often implemented as separate tasks, which are then invoked from the workflow. We argue that it should be possible to do these things in the workflow language itself.

In catering for additional language capabilities, we must be careful to retain the advantages that existing workflow languages already provide. Support for expressive and efficient programming logic can easily be obtained using general-purpose programming languages like C and Java, but these lack features required for workflows, such as implicit parallelism, integrated service access, and a data-oriented programming model. The challenge is in *combining* both expressive language constructs and the features required for workflows.

In addition to language features, we also address approaches for executing workflows across a distributed collection of machines. These approaches include *orchestration*, where a single client communicates with services in a client-server fashion, and *choreography*, where the workflow engine runs in a distributed manner, enabling direct communication between service hosts in a peer-to-peer fashion. We treat the choice of execution mode as an implementation issue, which is abstracted away from programmers by the language implementation.

Although our implementation does not address fault tolerance, the model itself leaves open the possibility that this could be addressed in the future. The side effect free nature of the programming model permits service calls to be retried or scheduled to other machines, and parts of the graph which have been lost due to machine crashes to be recovered by re-evaluating the expressions from which they were derived.

The programming language we present in this chapter is designed as a generic notation for expressing workflows. While it is possible to program in this language directly, it can also be used as a target for compilers of other, higher-level languages. Our approach is therefore applicable to the implementation of a range of different workflow languages that fit within the context of data-oriented processing.

3.1 Overview of our approach

Most workflow systems designed for e-Science applications are based on a *data-oriented* programming model, where tasks invoked by the workflow are connected together based on their data dependencies. This approach has the advantage that it possible for the workflow engine to analyse these dependencies and automatically determine which tasks can potentially be invoked in parallel. In achieving this however, existing workflow engines have sacrificed the ability to express a wide range of computational structures, particularly the implementation of complex logic directly within the workflow. Thus, existing workflow languages can only be used to instruct the workflow engine which tasks to execute, and complex logic must be implemented within external tasks.

The data-oriented programming models on which these languages are based are closely related to *functional programming*, where programs are expressed as side effect free transformations from input to output. Functional programming languages deal not with state manipulation, but rather the computation of results based solely on given inputs. The field of functional programming has been around for much longer than that of scientific workflows, and there are very well-established principles that provide the ability to implement languages which have a very simple programming model, yet are suitable for implementing a wide range of programs.

The core idea of this thesis is to take existing ideas from the functional programming community and apply them in the context of workflows. It is not sufficient to simply take an existing functional language implementation and write a workflow in it however, because existing functional languages do not include the capabilities of automatic parallelisation and integrated service access that are considered so important for workflows. Instead, we present the design and implementation of a new workflow system, which is based on established techniques for implementing functional programming languages, but adapts these techniques to cater for workflow execution.

There are three key aspects of the solution we have developed in this project. The first aspect is the basic programming model, which uses a small functional language we refer to as ELC (*extended lambda calculus*) to specify workflows, including the accesses that are made to remote services, as well as intermediate computation. The second aspect is *NReduce*, a distributed virtual machine which implements this programming language, and provides support for efficient execution, automatic parallelisation, access to network connections, and distributed execution across a set of nodes. The third aspect is an implementation of a subset of *XQuery*, a high-level functional language that is designed for processing XML, which we have adapted to support web services.

XQuery has the advantage of operating at a high level of abstraction, providing native language constructs for accessing, querying, transforming, and constructing data in XML format. Although it is possible to perform these tasks directly in ELC, doing so is more complex, as the programmer must work with a lower-level view of data in terms of cons lists and primitive data types. The availability of WSDL-based interface definitions for web services also makes it possible to automatically generate service access functions, which would otherwise have to be coded manually. XQuery is thus a superior choice to ELC for the development of workflows involving web services. Our XQuery implementation, which is discussed in Chapter 5, consists of a compiler which translates from XQuery to ELC, and a collection of support modules which provide functionality used by the generated code.

Within the context of NReduce, we explore implementation techniques that can be used to achieve parallel execution in two key contexts. The first is *orchestration*, a mode of execution in which the workflow is coordinated centrally by a single node, which sends off requests to remote machines in order to invoke services. The second is *choreography*, in which the virtual machine runs in a distributed manner across a set of nodes, each of which cooperates with the others to coordinate execution of the workflow. In the latter case, the services reside on the same set of nodes that host the workflow engine, and data can be transferred directly from producers to consumers, without incurring the additional costs involved with sending data via a central node.

3.2 Workflow language concepts

While considerable variation exists in the capabilities of different data-oriented workflow systems, they are all largely based on a common set of ideas, which we describe here. In Section 3.6, we explain how our own programming model maps to these concepts.

- **Tasks** are units of work that are used within a workflow. Each task takes a set of values as input, and, when executed, produces another set of values as output. A task can thus be considered equivalent to a function. Tasks may take many different forms, for example: jobs submitted to batch queuing systems, services invoked via RPC protocols, built-in functions provided by the workflow language, sub-workflows, and code fragments written in embedded scripting languages.

- **Data dependencies** are uni-directional relationships between tasks, each indicating that the input of one task comes from the output of another. During execution, these dependencies are used to determine which data to supply as input when executing a task, and also the relative order of execution of the tasks. In a data-oriented workflow, the data dependencies determine the ordering such that any data dependency from task A to task B indicates that A must complete execution before B may begin.

- **Control dependencies** indicate a constraint on the relative order of execution of two tasks, without implying data transfer. They are only necessary for tasks with side effects, where data dependencies alone are not sufficient to determine the ordering. If task A modifies some shared state that is also accessed by task B, and task B's correct execution depends on the state modification of A having been completed, then a control dependency can be added to the workflow to ensure that A must complete execution before B begins.

- **Parallelism** is possible when executing tasks due to the fact that the dependencies are made explicit in the structure of the workflow. Any set of tasks which are not directly or indirectly dependent on each other may execute in parallel. The workflow engine uses the control and data dependencies to determine which tasks may be run in parallel, without requiring the programmer to specify the parallelism explicitly, or manually deal with thread synchronisation. This automatic detection of parallelism is one of the most attractive properties of data-oriented workflow systems.

- **Conditional branching and iteration** play the same role in workflow languages as they do in other programming languages. A conditional branch chooses which part of the workflow will be subsequently executed based on a conditional test. The conditional test is performed by executing a task which takes the necessary information as input, and returns a boolean value. Iteration enables part of the workflow to be repeated multiple times, such that in each iteration, some or all of the tasks within the loop body are invoked with different input values. Many workflow languages only support sequential iteration, due to each iteration of a loop being triggered by a data or control dependency connected to the end of the loop.

- **Sub-workflows** can be used within other workflows, in a similar manner to user-defined functions. They aid in the construction of complex workflows by providing a means by which the programmer can abstract parts of the functionality into different sub-workflows, which are then invoked as tasks from other workflows. Sub-workflows may either be *named*, enabling them to be used from multiple locations, or *anonymous*, in which they are simply a grouping of operations into a single unit to facilitate viewing and editing. Sub-workflows can often be nested arbitrarily, though recursion is rarely supported.

- **Abstract workflows** are those in which the basic structure is defined, but some details are left unspecified. Usually, these details relate to the components or services used to carry out particular tasks. These contrast with *concrete workflows*, in which all of the information necessary for execution is available. An abstract workflow may be *instantiated* into a concrete workflow by a process in which the missing information is either inferred automatically, or supplied by a user. An important benefit of abstract workflows is reuse — the same workflow can be used with different data sets or collections of services without having to be re-written.

- **Embedded scripting languages** are supported by some workflow systems as a means of enabling the programmer to express computational logic that cannot be implemented in the workflow language itself, due to the limited expressiveness of the workflow language. Programmers can use these scripting languages to implement tasks involving things like data manipulation or complex application logic. Workflow programming often consists of development in two separate languages — the workflow language and an embedded scripting language.

- **Representation** of a workflow usually consists of a graph, in which nodes correspond to tasks, and edges correspond to dependencies. This is useful for both execution and editing. Parallel execution utilises the dependency information to determine which tasks can be safely executed in parallel. Editing is usually performed by graphical tools designed for end-users who lack extensive programming expertise, and provides an intuitive way to visualise the structure of a workflow. The syntax used for serialising these graphs to files usually consists of a set of statements specifying which nodes are present, and another set of statements specifying the edges between nodes.

3.3 Challenges addressed

In this section, we discuss what we believe to be the most important challenges regarding the programming models of workflow languages. While many of these requirements are already met by mainstream imperative and object-oriented languages, these languages lack important capabilities of workflow languages, such as automatic parallelisation and integrated service access. The problem we address is that of successfully *combining* all of these features into the one language, so that workflow developers can have a wide range of programming language features available to them without sacrificing ease of use.

3.3.1 Expressiveness

The *expressiveness* of a language refers to the range of possible programming logic that can be implemented in it. General-purpose programming languages like Java, C/C++, and Python possess a very high degree of expressiveness, while domain-specific languages such as database query languages and spreadsheet formulae have a very limited range of expressiveness. Workflow languages fit somewhere in between — they can be used to define a wide range of high-level processes based on composition of existing tasks, but lack the full capabilities of a general-purpose programming language.

Many languages have been designed with a deliberately limited degree of expressiveness, in order to open up opportunities for applying certain types of optimisations. For example, the simple programming models provided by workflow languages, in which data dependences between tasks are explicit, enable workflow engines to automatically determine when tasks can be safely executed in parallel.

It is our belief, however, that existing workflow languages sacrifice expressiveness to a greater degree than necessary. Application logic is often difficult to implement as part of a workflow, and must therefore be delegated to external tasks. We are therefore interested in exploring how increased language capabilities can be offered for defining workflows, while still maintaining the common benefits of existing workflow languages. Three important aspects of expressiveness we consider are *control flow constructs*, *abstraction mechanisms*, and *data manipulation*.

3.3.2 Control flow constructs

We use the term *control flow* to refer to the parts of a program that determine what computation is to be performed, based on various conditions. Examples of control flow constructs include conditional statements, iteration, function calls, and recursion. In a data-oriented language, control flow does not necessarily dictate the exact order in which execution will occur, but rather determines which expressions will be evaluated in order to compute the final result of a program.

Most scientific workflow languages provide limited forms of control flow, such as conditional statements and sequential iteration. However, these constructs are often restricted to certain types of usage, without exhibiting the full range of flexibility of their counterparts in general-purpose programming languages. For example, data structure traversal involves both conditional tests and either iteration or recursion, but is usually not something that can be expressed in a workflow language. Additionally, the use of expressions involving arithmetic, relational, and boolean operators is often awkward, particularly if the workflow language does not provide an embedded expression syntax in which to specify the conditions for branching or iteration.

Recursion is a common programming technique used with functional languages, but is generally not supported by workflow languages. In order to perform iteration in a side effect free functional language, it is necessary to make recursive calls to a function in which the parameters take on new values, rather than modifying existing state representing a loop counter. Due to our focus on the application of functional programming ideas to workflows, recursion is an important feature for us to support.

3.3.3 Abstraction mechanisms

Complex programs are developed as a set of individual parts, each of which implements some portion of the program's functionality, and is composed together with other parts to comprise the whole program. In order to support such structured development, a language must provide constructs which enable these parts to be defined and composed. These different parts are often organised into different levels of abstraction, whereby low-level parts deal with more specific implementation details of the application, while high-level parts specify the overall logic of the application, without addressing specifics.

Existing workflow languages support the expression of the high-level coordination aspects of an application (known as *programming in the large* [87]), but not low-level implementation details (*programming in the small*). Workflow languages are designed based on the assumption that all implementation details will be addressed within tasks invoked by the workflow. These tasks are either external pieces of software, or built-in operations provided by the workflow language, both of which must be developed using separate languages to the workflow itself.

The problem with this approach is that it places an unnecessary barrier between the levels of abstraction within a program. While simple workflows can be constructed using a restricted set of language constructs, there are often situations in which something more complex needs to be achieved — for example, parsing a string to produce a tree structure, or sorting a list based on a user-defined comparison function. When this is the case, the workflow developer is forced to switch to a different programming language, such as the one in which the workflow engine itself is implemented. They must use this language to implement a separate task which is then invoked by the workflow.

Our belief is that instead, the workflow language itself should support *multiple levels of abstraction*, whereby both the high-level aspects of a workflow, such as overall application logic, and low-level aspects, such as service access protocols, can be implemented in the same language. The provision of suitable abstraction mechanisms, as well as appropriate language constructs for control flow and data manipulation, aids the development of a wider range of programming logic within the workflow language. This does not require a complex programming model — rather, it simply requires a programming model that provides a *suitably general* set of abstraction mechanisms.

3.3.4 Data manipulation

Workflow languages typically treat data as *opaque* — that is, they allow data values to be passed around between services, but do not provide built-in language constructs for examining or manipulating this data. From a high-level point of view, this seems to make sense, since workflow developers are primarily concerned with the composition of tasks. However, there are many circumstances in which it is necessary to perform intermediate manipulation of data, such as converting numbers between different units, or extracting fields from a record structure.

In the absence of any built-in data manipulation operations, workflow developers are often forced to switch to a different programming language, and implement a separate component which performs the necessary processing, and is then used as a task within the workflow. For example, it may be necessary to extract all values of a certain field in a collection of records, and concatenate them together into a string — something which is impossible in most workflow languages. Having to switch to a different language to carry out such a simple activity is very inconvenient for programmers, and detracts from the ease of use that workflow languages are so often promoted as providing. We argue that a more sensible approach is to make the workflow language expressive enough for this logic to be implemented directly, so that the developer only has to deal with a single language.

In most programming languages, data values fall into one of two main categories: *atomic values* and *structured data*. Examples of atomic values are integers, floating point numbers, and boolean values. They must be manipulated by built-in language primitives such as arithmetic and logical operators. Examples of data structures are records, tuples, lists, and trees. They are manipulated by language constructs that enable the programmer to access particular fields of a structure, and create new objects which contain references to other values or data structures. It is our belief that workflow languages should adopt facilities from existing programming languages that enable data to be manipulated in this manner.

3.3.5 Implicit parallelism

One of the most important advantages of data-oriented workflow languages is that they abstract away details of parallel execution from the programmer. This is desirable because a) workflows invoke tasks on external machines, and it is advantageous to use these in parallel, and b) workflow languages are designed to be easy to use for those without advanced programming skills. Most other parallel programming technologies use an imperative model of execution, in which complex and error-prone operations such as thread synchronisation and message passing must be explicitly coded by the programmer.

Existing parallel functional languages successfully abstract away low-level thread interaction issues due to their side effect free nature, but do not completely avoid the need for explicit control of parallelism. Previous work in this area has focused solely on cases in which the whole application is implemented in the language in question, and is thus extremely sensitive to the per-operation overheads involved with managing parallelism. A consensus has been reached that in these situations, completely automatic parallelisation is infeasible [166].

Workflows differ from the types of programs studied previously in the context of parallel functional programming, in that most of their compute-intensive work is performed by external tasks. The workflow language itself does not have to be highly efficient, because only a small portion of the total execution time is spent carrying out the coordination logic. Automatic parallelisation is already widely supported by workflow languages, even though it imposes overheads, because the number of operations in most workflows is relatively small. However as the complexity of the logic expressed in a workflow increases, it becomes more important to keep these overheads low.

Implicit parallelism makes the other goals in this section harder to achieve. Providing additional language constructs for control flow and data manipulation using an imperative approach is straightforward if one does not desire automatic parallelisation. However, *combining* these features with parallelism in a way that is suitably efficient is a significant challenge. Care must be taken in designing the execution mechanism to ensure that the overheads incurred by increased amounts of computation are low enough to avoid significantly impacting overall performance.

3.3.6 Service access

In this project, we consider workflows in which all tasks correspond to network-accessible services invoked via a request/response message exchange. In existing workflow languages, service operations are exposed as atomic operations, with the implementation logic used to encode parameters and communicate with the service residing internally within the workflow engine. This makes the service access mechanisms difficult to customise — if a developer wants to access a different type of service for which no built-in support is provided, they must modify the workflow engine. We argue that a more flexible approach is to enable the service access logic to be implemented in user code, so that it is easier to customise.

In choosing a suitable set of abstractions upon which service access mechanisms can be implemented, it is important to consider the two key aspects involved with communicating over a network: *protocol syntax* and *data transfer mechanisms*. Protocol syntax refers to the way in which information is encoded into a sequence of bytes that can be transferred over the network. This is appropriate to handle within a data-oriented programming model, as it can be implemented using side effect free functions which transform data from the internal language representation to the on-the-wire format, and vice-versa.

Data transfer mechanisms refer to the way in which the program interacts with the operating system to organise transfer of the encoded data over the network. This involves making system calls to establish network connections, as well as to read and write data. Additionally, if multiple services are being invoked in parallel, then either multi-threading or an event loop must be used by the program to handle the arrival of data from different connections at different times. This logic does not fit cleanly within a data-oriented programming model, and is an entirely separate issue to data encoding. We therefore argue that this aspect of service access is best handled internally by the workflow engine.

Some programming languages are not well-suited to dealing with data at the level of byte streams, because they represent data at a higher level of abstraction. An example of such a language is XQuery, in which all data is represented in XML format. For these languages, it is less practical to provide the ability to implement service access mechanisms in user code, because the language itself lacks the necessary primitives for encoding data. However, for languages that operate at a lower level of abstraction, and provide built-in operations for dealing with arrays of bytes, the ability to customise service mechanisms is more practical to provide. In our particular case, we provide this ability within ELC.

3.3.7 Distributed execution

In addition to the above challenges, we also address the issue of execution strategy. Workflows are inherently distributed in nature, so the execution mechanism must address the interaction between multiple machines in a suitable manner. The two main approaches to this are *orchestration* and *choreography*, which we discuss in the following section. The way in which we support these two modes of operation makes the choice between them an entirely separate concern to the programming model, enabling programmers to focus on *what* their workflow does, rather than *how* it does it.

3.4 Orchestration and choreography

Orchestration and choreography, introduced in Section 2.3.4, are two different ways of executing a workflow. In the first, the client runs on a single node, and makes calls out to services by exchanging request and response messages with services over the network. With choreography, the services interact directly with each other, under the general direction of the workflow. In both cases, the workflow specified by the programmer determines which services are invoked and how the data is exchanged between them; the choice of orchestration vs. choreography is one of deployment and execution strategy.

3.4.1 Orchestration

Orchestration involves all control decisions regarding workflow execution being made by a client[1]. Each time a service is invoked, the client sends a request across the network containing the necessary parameters, and subsequently receives a response message. None of the services interact with each other directly; any data that needs to be passed from the result of one service to another always goes through the client. This scenario is shown in Figure 3.1.

Parallelism may be achieved in this scenario by having the client make multiple concurrent requests to different services. The way in which this works is much like a web browser that has multiple connections to different web servers to download different elements of a page, such as images and style sheets. The client-side execution model must support the notion of asynchronous requests, whereby the program or workflow does not have to block while it has one request open, and can instead continue on with making other requests.

A key advantage of orchestration is that because everything is controlled from a central point, it is relatively straightforward to implement. Each individual service needs to communicate with only one other node — the client — and the request/response pattern is used for all interaction. This pattern is the most common way in which existing services are implemented, making it the most widely applicable. No special workflow logic is needed on the machines hosting the services, because invocations from the workflow engine appear exactly the same as calls made by any other type of client.

One disadvantage of orchestration, however, is that it can often result in more data being transferred across the network than is necessary. Whenever the output of one service is

[1]In some cases, the client software responsible for performing orchestration may actually execute on a different computer to that on which the user interface resides.

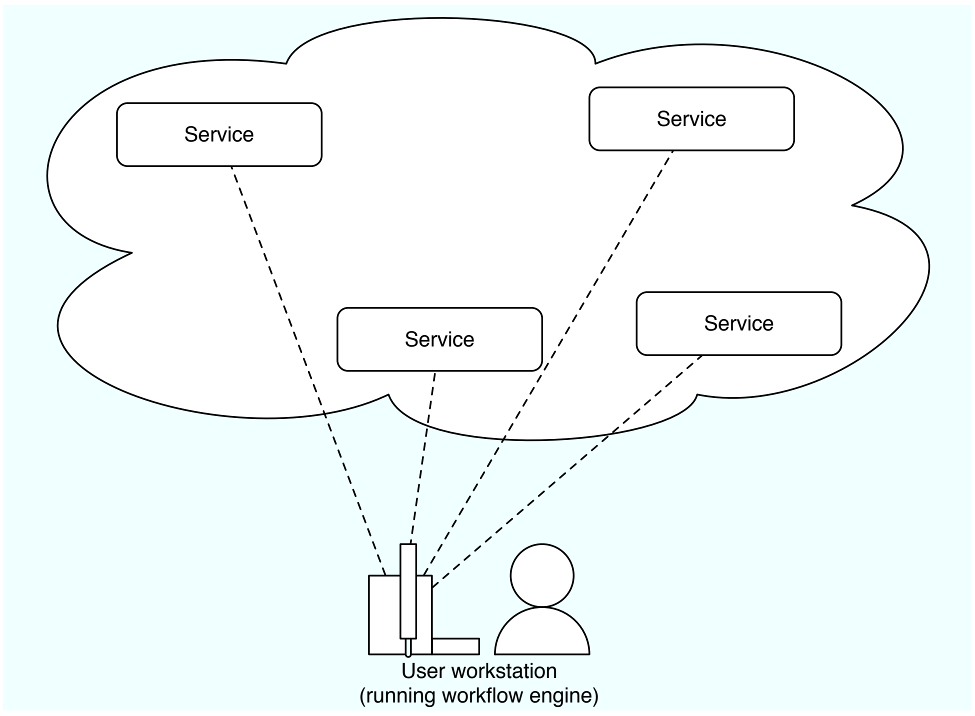


Figure 3.1: Orchestration

used as the input to another, the data must always go through a central node, rather than directly between the services. If the two services being invoked are located on the same host, the data must be sent to the client and back again, instead of just transferred directly in memory between the two services. For this reason, choreography can be an attractive alternative.

3.4.2 Choreography

With choreography, the services involved with a workflow interact directly with each other. When the output of one service needs to be passed to another, the data can be sent directly between the two machines on which the source and destination service reside, without going through a central node. For workflows that involve large volumes of data, and services distributed over a wide-area network such as the Internet, this can lead to substantial performance benefits, because less bandwidth is used.

A significant challenge in implementing this is that in addition to its own application logic, each service must also contain coordination logic relating to its participation within the workflow. When a service produces a result, it needs to know where to send the result, and the recipient needs to know what to do with it. Additionally, any intermediate computations included within the workflow must somehow be be supported. By itself, the client/server model used by most services is insufficient to support this mode of operation, since it simply involves having each service send its results directly back to the client which invoked it.

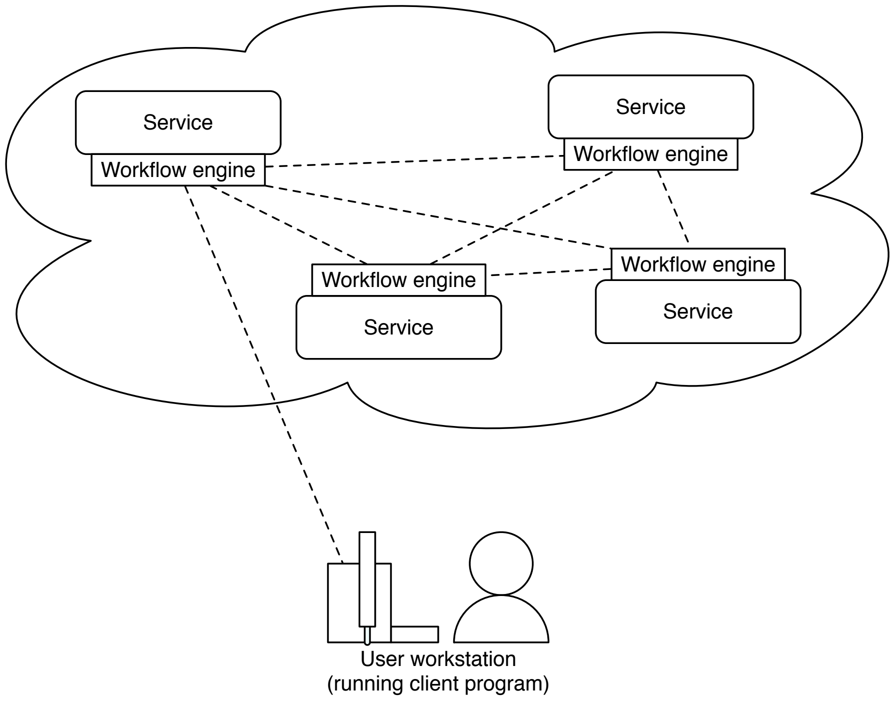


Figure 3.2: Choreography using co-hosted workflow engine nodes

It is possible to indirectly choreograph such services by placing additional software on each service host which acts as a wrapper around the service, invoking it in the usual manner, but handling the result appropriately afterwards. In our model, we achieve this by implementing our workflow engine in a distributed manner, such that instances of the engine are deployed on each machine, and these interact with each other to execute the workflow. The services themselves are entirely unaware that they are participating in choreography, as they perform all interaction using the standard request/response mechanism, with the local instance of the workflow engine acting as the client.

Figure 3.2 shows the way in which our workflow engine and a set of services would be configured to achieve choreography. Each machine runs both an instance of the workflow engine and one or more services. The user's workstation runs a client program that communicates with the workflow engine to launch workflows and retrieve results. Any time a particular service needs to be invoked, the request is made from the instance of the workflow engine that runs on that node, and the request/response messages are exchanged over the local network interface. Any data transfers needed to pass results to other services occur between the nodes of the workflow engine.

This approach has the limitation that it requires the ability to deploy the engine onto the hosts on which services are accessed. This is not always possible, due to administrative restrictions that are generally present on publicly accessible servers. However, it is a practical approach in controlled environments where a single organisation or group of collaborating organisations is willing to deploy the workflow engine on their servers in order to gain the performance benefits made possible by choreography.

Regarding the workflow model, an important concern is how to determine the message exchanges necessary to achieve choreography. While it would be possible to expose this concern as part of the programming model, doing so would make workflow development harder for programmers by mixing functional with non-functional concerns. We instead use an implicit approach in which all data transfers are determined automatically and dynamically by the runtime system, using existing techniques for distributed memory parallel graph reduction. This is purely an implementation issue, which we defer discussion of to Section 4.7. Because of this approach, the programming model we define in the following section contains no explicit notion of choreography, enabling workflows to be defined purely in terms of their functionality.

3.5 Extended lambda calculus

A major goal of this project is the exploration of a programming model for data-oriented workflows which addresses the challenges described in Section 3.3. The dataflow-based models and coarse-grained primitives of existing workflow systems do not provide sufficiently flexible ways of expressing data processing and application logic. We argue that by adopting principles from the world of functional programming, and implementing these ideas in a suitable manner, workflow languages can be made just as flexible as mainstream scripting languages, while retaining the important benefits of implicit parallelism and integrated access to services.

Functional programming languages are based on *lambda calculus* [67, 68, 310], an abstract, Turing-complete model of computation which defines a very simple set of rules by which expressions are evaluated. Despite its apparent simplicity, this model permits an extremely wide range of programming techniques to be used. This is due to its generality, and focus on evaluation semantics rather than practical features. Lambda calculus has been at the foundation of functional programming research for decades, and its properties are well understood within the programming language community. In this thesis, we put forward the argument that it is ideally suited as a model for defining and evaluating workflows.

The following discussion assumes that the reader already has a solid understanding of functional programming and lambda calculus. For space reasons, it is not practical to include a full explanation of these topics; moreover, they have already been covered extensively in other literature. For those new to the area, we recommend the book *Structure and Interpretation of Computer Programs* by Abelson and Sussman [2]. This book is based on the functional language *Scheme* [184], which is very similar to ELC. Another excellent resource is the book *An Introduction to Functional Programming Through Lambda Calculus* by Michaelson [231]; we provide a brief review of this material in Section 3.5.1 and Appendix A.

By itself, lambda calculus only defines an expression syntax and evaluation rules, and lacks any notion of built-in functions or data types. In order to use the model in any practical situation, one must define extensions that dictate what primitive functions are available, and what types of data can be manipulated. Additionally, it is common to define additional syntactic constructs to simplify certain programming tasks, such as defining recursive functions. Most pure functional languages can be considered extensions to lambda calculus.

In this project, we have developed a small functional language we refer to simply as ELC (*extended lambda calculus*). This language defines a small set of primitive functions and data types, to facilitate the practical aspects of workflow definition. Additionally, it provides built-in functionality to access remote services, by treating all network connections as functions which transform streams of input data to streams of output data. The built-in functions and data types, parallel execution semantics, and network access mechanisms defined in this section together comprise our programming model.

The specifics of the language given in this section are not critical to our core argument about the utility of functional programming for developing data-oriented workflows. Considerable variation in syntax, data types, and built-in functions is possible while still following our basic philosophy of using lambda calculus as a model for workflows. It is our hope that this philosophy will be applied in the context of other workflow engines developed in the future.

This section only deals with ELC itself, defining the syntax and semantics of the language. For a discussion of how ELC can be used to develop workflows, see Section 3.6.

3.5.1 Lambda calculus

Expressions in lambda calculus consist of *variables*, *lambda abstractions*, and *function applications*. A variable may either be *bound*, corresponding to a function parameter, or *free*, in which case its meaning depends on the context in which the expression is being evaluated (e.g. the set of built-in functions and constants). A lambda abstraction defines a function of one argument. A function application corresponds to an application of a lambda abstraction to a value. Functions of multiple arguments may be defined by nesting lambda abstractions, such that a function of $n > 1$ arguments returns a function of $n - 1$ arguments which may subsequently be applied to the relevant values. Lambda calculus has no explicit concept of operators; every operation that can be performed is modeled as a function. Pure lambda calculus does not define any built-in functions or constants, but programming languages based on the model always specify some set of primitives such as numbers and arithmetic functions.

All computation in lambda calculus occurs according to a single rule: β-reduction. This involves converting an expression of the form $(\lambda x.E)\, M$ into $E_{[M/x]}$, the expression resulting from substituting all instances of the variable x in the expression E with the argument M. This rule is equivalent to a function call, where the body of a function is evaluated in an environment in which variables corresponding to formal parameters are associated with the values passed in as actual parameters. Due to the Church-Rosser theorem [69], which states that the result of evaluation is independent of the order in which expressions are reduced, multiple expressions which are independent of each other may safely be reduced in parallel.

3.5.2 ELC

The ELC language adds the following features, which are largely similar to those supported by most other functional languages:

- **Numbers.** All numbers are represented as 64-bit double precision floating point numbers. This provides flexibility without requiring the language implementation to deal with conversion between different numeric types, or provide multiple versions of each operator for different data types.
- **Arithmetic and relational operators.** These include addition, subtraction, multiplication, division, and modulus, as well as equality comparisons and less than/-greater than operators.
- **Cons pairs.** Like other functional languages, ELC includes native support for linked lists, built out of a series of cons pairs [221]. Each pair has a *head* and a *tail*; the head is used to store a data item, and the tail is used to point to the next part of the list, or to the special value *nil*, which indicates the end of the list.
- **Letrec expressions.** These permit symbols to be bound to expressions, optionally in a mutually recursive manner. This enables sharing to be expressed, such that multiple uses of a symbol within an expression all point to the same instance of the bound expression. Letrec bindings differ from variable initialisation in imperative languages in that they are not necessarily executed in a sequential order.
- **Top-level function definitions.** An ELC program consists of a series of function definitions, each of which may have zero or more arguments and a body. In pure lambda calculus, functions can only be expressed as lambda abstractions, which may be subsequently bound to a symbol by supplying the lambda abstraction as an argument to a function. As in other functional languages, our top-level function definitions allow globally-accessible identifiers to be bound to functions, and used within any expression.
- **Strings.** These are technically not part of the language semantics, but are supported natively by the parser. All strings in ELC are represented as cons lists, where each item is a single character in the string, represented by its numeric character code. This permits strings to be manipulated in exactly the same manner as other types of lists. Standard list functions, such as concatenation and subsequence extraction, can be used with strings as well.
- **Library functions.** A number of commonly-used functions are defined in a *prelude* file, and made available to all ELC programs. These include functions like `map`, `filter`, and `foldr`, which are commonly used in functional programming. The library functions do not form part of the core language, and could theoretically be provided directly in user programs. However, these library functions are provided in the prelude file for convenience, and to enable certain list functions to take advantage of implementation-specific optimisations.

- **Modules.** In order to facilitate code reuse, ELC permits the source code for a program to be split across multiple files. Each file is called a *module*, and is given a namespace based on its filename. A program can import other modules by declaring them at the top of the file. This is similar to an include mechanism, except that all functions defined in the included file are prefixed with the name of the file, followed by a double colon (`::`), to avoid name clashes. We make use of the module functionality in our XQuery implementation, as described in Section 5.5.1.

  For example, a program containing the following import statement:

  ```
  import xml
  ```

  can access functions defined in the file `xml.elc` by using the prefix `xml::` — for example, `xml::mkitem` and `xml::item_type`.

Programmers familiar with Scheme will find ELC very easy to pick up, as the two languages have a great deal in common. The main differences are as follows: ELC uses a slightly different syntax for what Scheme calls *special forms*, such as letrec expressions, lambda abstractions, top-level function definitions, and module imports, although other parts of the code are written as S-expressions. All functions in ELC take a fixed number of arguments, whereas in Scheme they can take a variable number of arguments. Some of the syntactic sugar of Scheme, such as quoted lists and `cond` expressions, are not included in ELC, since these things can be expressed in terms of lower-level primitives, such as calls to `cons` and sequences of `if` statements. Finally, ELC prohibits side effects; this restriction is necessary to make automatic parallelisation feasible.

An important design philosophy we adopted from Scheme is that "Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary" [184]. We believe that this approach is just as relevant for workflow languages as it is for other types of programming languages. ELC only provides a very small set of built-in functions and data types, and requires the programmer to define any higher-level functionality that they need. The reason why we followed this approach was to simplify our implementation.

In particular, we have applied this philosophy to the way in which we permit services to be accessed from within a workflow. Instead of imposing the use of particular protocols and data formats such as SOAP and XML, ELC provides a highly generic mechanism for manipulating data as strings and byte streams, and transferring these to and from services directly over TCP connections, a feature we introduce in Section 3.5.8. This approach is designed to enable implementations of higher-level workflow languages which use ELC as a target compilation language to include their own implementations of particular data formats and protocols in terms of ELC, a topic we explore with our XQuery implementation in Chapter 5.

3.5.3 Built-in functions

The most important extensions that ELC adds to lambda calculus are a set of built-in functions which can be used to manipulate atomic values and data structures. These

Basic operations	
`+, -, *, /, %`	Arithmetic functions
`==, !=, <, <=, >, >=`	Relational operators
`<<, >>, &, |, ^, ~`	Bit operations
`and, or, not`	Boolean functions
`if cond t f`	Returns `t` or `f` depending on the value of `cond`
Cons pairs	
`cons h t`	Creates a cons pair with head `h` and tail `t`
`head p`	Returns the head of cons pair `p`
`tail p`	Returns the tail of cons pair `p`
Lists and strings	
`len list`	Computes the length of a list
`item n list`	Returns the `n`th item of `list`
`skip n list`	Skips past `n` elements of `list` (like `n` calls to `tail`)
`prefix n list`	Returns the first `n` elements of `list`
`sub start count list`	Obtains a subsequence of a list
`++ first second`	Returns the concatenation of `first` and `second`
`map f list`	Returns a list containing the results of applying the function `f` to each element in `list`
`filter f list`	Returns only the elements of `list` for which `f` evaluates to true
`foldl f base list`	Performs a left-fold on the values in `list` using the function `f`, with `base` as the initial value
`foldr f base list`	Performs a right-fold on the values in `list` using the function `f`, with `base` as the initial value
`ntos n`	Converts a number to a string
`ston s`	Converts a string to a number
`strcmp a b`	Compares two strings
`streq a b`	Checks two strings for equality
Network access	
`connect host port data`	Connects to a server, sends it `data`, and returns the response

Table 3.1: Built-in functions

enable ELC to be used as a realistic programming language, taking advantage of hardware support for arithmetic and logical operations, which in pure lambda calculus have to be expressed using non-intuitive techniques such as Church numerals [68].

Table 3.1 contains a summary of the functions that are available in ELC. The first two categories are implemented natively as C functions within the interpreter. The arithmetic and relational operators operate according to the standard semantics for double-precision floating point numbers, and the boolean functions are equivalent to their counterparts found in other programming languages. True and false values are distinguished as follows: nil represents false, and any other non-nil value is interpreted as true.

Conditionals are modeled as a function called `if`, which takes parameters corresponding to the conditional expression, the true branch, and the false branch. `if` is simply a function, rather than an explicit language construct. Its only special property is that its first argument is strict, while its second and third arguments are non-strict, in order to avoid evaluating branches whose values will not be used, particularly within recursive functions.

The `cons`, `head`, and `tail` functions are used for constructing and accessing cons pairs. `head` and `tail` correspond to `car` and `cdr` in Scheme, and have the same semantics. Fundamentally, these three functions are all that is needed to operate on lists, however for convenience and efficiency a number of other list processing functions are also provided. While `cons`, `head`, and `tail` are implemented directly within the interpreter, the other list functions, such as `item` and `len`, are implemented in user code, and made available to all programs by virtue of their inclusion in the standard prelude module.

From a programmer's perspective, the list functions can be thought of as operating directly on linked lists of cons cells. In fact, one could easily define all of these functions in terms of `cons`, `head`, and `tail`. The only difference in the implementation of these functions is that in certain cases they are able to make use of an optimised list representation, described in Section 4.6, in which elements are stored in arrays. This optimisation is invisible to the programmer, and is not technically part of the programming model proper. We made the deliberate choice to avoid exposing arrays as an explicit language feature in order to keep the programming model as simple as possible.

All strings are represented as lists of numbers corresponding to character codes, and thus the list functions can be used for manipulating strings as well as other types of lists. A few special functions are provided for dealing with strings: `ntos` and `ston` convert between numbers and strings, and `strcmp` and `streq` perform comparisons between strings. These functions could be implemented in ELC using the other built-in functions described above, but the versions provided in the prelude file are able to make use of the optimised list representation where possible.

The `connect` function, used for network connections, is described in Section 3.5.8.

```
Program     = Import* Definition+

Import      = "import" SYMBOL

Definition  = SYMBOL Argument* "=" Expr

Argument    = SYMBOL | "!" SYMBOL | "@" SYMBOL

Expr        = NUMBER | STRING | SYMBOL | "nil"
              | "(" Expr+ ")"
              | "(" Lambda+ Expr+ ")"
              | "(" "letrec" Letrec+ "in" Expr ")"

Lambda      = "\" SYMBOL "."

Letrec      = SYMBOL SYMBOL* "=" Expr
```

Figure 3.3: ELC syntax

3.5.4 Syntax

The formal grammar of ELC is given in Figure 3.3. The top-level production is *Program*, which contains a (possibly empty) sequence of `import` statements, followed by one or more function definitions. Each function may have any number of arguments, including zero. A function of zero arguments is equivalent to a constant, since all functions in ELC are referentially transparent. Arguments may optionally be annotated with `!` to indicate strict evaluation, or `@` to indicate lazy evaluation.

Each expression may either be a constant, bracketed expression, lambda abstraction, or letrec expression. Constants are either numbers, strings, symbols, or the special value *nil*. A sequence of *Expr* productions inside brackets corresponds to a function call, in which the first expression is the function, and the remaining expressions are arguments. A lambda abstraction defines an anonymous in-line function; several of these may be nested in order to define a function of multiple arguments. The `letrec` keyword followed by a series of bindings, the `in` keyword, and a body defines an expression in which the specified symbols are bound to their corresponding expressions. Lambda abstractions are written using \ in place of the λ symbol, since the latter is not part of the ASCII character set. Additional symbols given before the `=` sign in letrec bindings are equivalent to specifying lambda arguments after the `=` sign.

In addition to following these syntactic rules, all programs are required to define a function called `main`, whose body forms the top-level expression that will begin evaluation when program execution starts. This function may optionally accept a single parameter corresponding to the list of arguments passed to the program on the command line.

```
countspaces str =
(if str
    (if (== (head str) ' ')
        (+ 1 (countspaces (tail str)))
        (countspaces (tail str)))
    0)

main args =
(if (== (len args) 0)
    "Please specify a string"
    (++ "String contains " (++ (ntos (countspaces (head args)))
                               " spaces\n")))
```

Figure 3.4: Counting spaces in a string

3.5.5 Programming example

Figure 3.4 shows an example of a simple ELC program which takes a string from the command line, and prints out the number of spaces in it. The `args` parameter to the `main` function is a list of strings, which is first checked to see if it contains one or more items. If this is not the case, the program will print the message "Please specify a string". Otherwise, it will make a call to the `countspaces` function, passing the first item in the argument list as the parameter. The numeric value returned by `countspaces` is converted to a string, and concatenated with two other strings to produce a result of the form "String contains *n* spaces\n". The string returned from `main` is printed to standard output by the runtime system.

The `countspaces` function uses recursion to step through the string character by character. At each position, it checks to see if the character at the current position is a space, and if so, adds one to the result of calling itself recursively to count the number of spaces in the rest of the string. If the current character is not a space, it simply makes a tail recursive call to itself. The base case is triggered when the end of the string is reached, in which case `str` will be nil, which is interpreted by the `if` function as meaning false.

3.5.6 Evaluation modes

ELC supports both strict and lazy evaluation. This enables us to explore the performance characteristics of workflows running in either mode, to determine the most suitable type of evaluation. Both have their advantages and disadvantages; strict evaluation makes it easier to detect parallelism, while lazy evaluation has the potential to allow large data sets to be processed in a streaming fashion. In Section 6.1, we explore the differences in behaviour of the two evaluation modes within the context of workflows.

Lazy evaluation is also supported because it is needed for creation of cyclic data structures. These are used in our XQuery implementation for representing trees of XML nodes, in which pointers are maintained both upwards and downwards in the tree. With strict evaluation, it is impossible to create two objects which refer to each other, since the lack of side effects prohibits the modification of an object after it has been created. With lazy evaluation, even in the absence of side effects, it is possible to create a cyclic reference by initialising a field of one object with a suspended evaluation referring to a second object, which in turn contains one or more references back to the first. Section 5.5.5 explains our use of this technique in more detail.

The evaluation mode can be set as a configuration parameter of the virtual machine, which defaults to strict evaluation. It can also be controlled at a more fine-grained level, by specifying whether certain parameters to a function should always be evaluated strictly, or always evaluated lazily. As mentioned in Section 3.5.4, strict evaluation is indicated by prefixing a parameter in a function definition with `!`, and lazy evaluation is indicated by prefixing the parameter with `@`. If either is specified, it will override the evaluation mode set in the configuration, but only for the relevant parameter of the function in question. This mechanism makes it possible to use either mode for the majority of code, while preserving strict or lazy evaluation for functions which need a specific mode.

3.5.7 Parallelism

Parallel evaluation of expressions in ELC is triggered as follows: Whenever a call is made to a function that has two or more strict arguments, all such arguments are *sparked*, marking them as candidates for parallel evaluation. The goal of parallelism in our case is not so much to exploit *internal* parallelism within the code, but rather to exploit *external* parallelism, in which multiple service calls may run at the same time.

There are two ways in which sparked expressions may be evaluated in parallel, depending on whether the virtual machine is running on a single node (orchestration) or across multiple nodes (choreography). In the first case, only a single processor is involved in execution of the code, so there is no internal parallelism in the sense of multiple expressions being reduced simultaneously. However, sparks are useful whenever the processor would otherwise become idle due to an expression performing a blocking I/O operation such as reading or writing data from a network connection. In such a case, the virtual machine switches to evaluating a different expression while the I/O operation is in progress.

When the virtual machine is running across multiple nodes, sparked expressions may be distributed from one node to another when a work request from an idle node arrives. This enables multiple nodes to each be evaluating different expressions. This is internal parallelism, in that it applies to the logic within the program itself. However, the focus is on distributing tasks across the nodes so that service calls can be made on all machines involved with the choreography.

The way in which parallelism is triggered depends on which evaluation mode is in use. With strict evaluation, functions are considered strict in all of their arguments, which means that all arguments can be evaluated in parallel. With lazy evaluation, a process of *strictness analysis* (discussed in Section 2.4.1) is used to determine which arguments a function will definitely evaluate, and only these arguments are eligible for parallel evaluation. Strictness analysis is necessarily conservative, in order to avoid unnecessarily triggering evaluation of expressions that may lead to non-termination. For this reason, there are fewer opportunities for parallel evaluation detected in the case of lazy evaluation. The implications of this difference are explored in Section 6.1. Strict evaluation is the default.

3.5.8 Network access

Since the category of workflows we are targeting involves coordination of services, an important consideration in the language design is the way in which service invocation is exposed. At a basic level, this requires the ability to establish network connections through which data can be sent to and received from services. In a side effect free programming model based on purely data-oriented semantics, this network interaction should be exposed in a manner which fits these semantics, rather than the more traditional approach of operations that manipulate state.

As explained in Section 2.4.6, a common assumption about I/O in functional languages is that it always involves side effects, and thus the language should explicitly ensure that all interaction with the outside world happens in a sequential order. This approach is incompatible with our requirement to be able to invoke multiple services in parallel, which fundamentally relies on the ability to have multiple asynchronous I/O operations in progress at the same time.

The key property of our target applications that allows us to avoid sequential I/O is that all services they invoke are side effect free. This is not enforced by our model or implementation, but is an assumption we make based on the intended usage scenarios. In making this assumption, we shift the responsibility on to the workflow developer to make sure that the services they are accessing are indeed free of side effects, at least in the sense that there is no state modification that would cause the result of the workflow to be dependent upon the order of execution. This is a common requirement of many parallel programming frameworks such as task farming, MapReduce [83], and other data-oriented workflow engines.

Our approach to providing network access to user code is to expose connections using *data streams*, each of which appears to the program as a cons list. Each connection has an *input stream* and an *output stream* associated with it, corresponding to the data that is sent and received across the network. The input stream is written to by having the runtime system traverse a cons list produced by the program, and send its contents over the connection. The output stream is read from by the program traversing the list and accessing its contents.

This way of exposing network connections to user code enables programs to deal purely with data, for both production and consumption. The state-based interaction involving explicit send/receive operations and blocking is decoupled from the programming model, and handled internally by the runtime system. Assuming that all services accessed by a

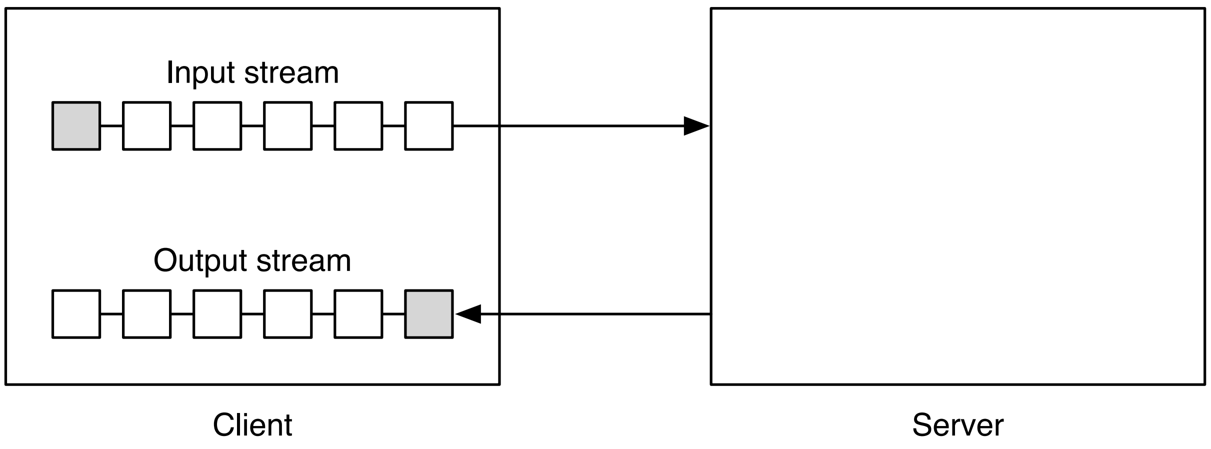


Figure 3.5: Input and output streams (lazy evaluation)

program are side effect free, this fits entirely within the data-oriented view of functional programming. Parallel service invocation is possible due to the ability to have multiple expressions under evaluation at the same time, each producing or consuming a separate data stream. If one expression blocks while trying to read or write data, others may continue. A programmer can thus invoke multiple services in parallel, without any manual use of parallel programming constructs. Section 4.5 explains how this abstraction is implemented.

A network connection is established by calling the `connect` function, which takes three parameters: the host, port number, and input data to be sent to the server. The result of a call to `connect` is the data received from the server in response to the request. An example use of this function is to make a HTTP GET request to retrieve the contents of a web page:

```
main = (connect "www.adelaide.edu.au" 80
                "GET /research/index.html HTTP/1.0\r\n\r\n")
```

In this example, the result of the call to `connect` will be printed out as the result of the program, in this case containing the response data received from the web server, including HTTP headers.

The exact nature of the way in which the input and output streams are processed depends on the evaluation mode in use. With strict evaluation, the whole input stream must first be evaluated before the connection is established and the data is sent to the server, and the complete response must arrive before any of it can be accessed by the client. With lazy evaluation, the input stream can be produced and sent incrementally, and the output stream can be parsed incrementally as data arrives.

Figure 3.5 shows an example of the lazy evaluation case. Only part of the input and output streams have been materialised, with the shaded cells representing the portions that are yet to be produced. This mode allows streaming processing, whereby data is processed as it arrives, and very large data streams can be handled using a constant amount of memory.

3.6 ELC as a workflow language

Whereas the focus of the previous section was on describing the syntax and semantics of ELC, we shall now discuss how it is useful as a workflow language, and describe its benefits over existing workflow languages. As discussed in Section 2.3.8, the latter have many restrictions which make it difficult to express certain types of programming logic within a workflow. As a language which is very similar to functional languages designed purely for computation, ELC provides a much more expressive programming model than existing workflow languages, while retaining the benefits. Our intention is to demonstrate that the restrictions of these languages are not fundamentally necessary, and that it is possible to achieve both ease of use and expressiveness within the one language.

In principal, it is possible to define a workflow in any general-purpose programming language. For example, one could write a workflow in C by calling socket functions provided by the operating system to communicate with services, and using threads or an event loop to coordinate parallel execution of multiple services. However, doing so requires the programmer to have substantial expertise in dealing with low-level implementation issues such as memory allocation, system calls, and thread synchronisation. Workflow languages are popular because they abstract away these details, and provide programming models which are much simpler to understand. However, with existing workflow languages, this simplicity comes at the cost of expressiveness. ELC is designed to be easy to use, *without* placing restrictions on what can be expressed within a workflow.

Our main motivation in choosing lambda calculus as an abstract workflow model is its flexibility. Existing workflow languages have various restrictions which prevent them from being used to express the full range of logic that one may wish to include in a workflow, necessitating the use of external "shim" services and embedded scripting languages. In contrast, lambda calculus is a Turing-complete model of computation, capable of expressing any possible programming logic. This alleviates the need for programmers to switch to a different language when they want to implement application logic or data transformation.

Lambda calculus has several properties which make it particularly well-suited for data-oriented workflows. Firstly, its lack of side effects leaves open the possibility of automatically detecting opportunities for parallel evaluation. Secondly, its model of producing a result based on input values without modifying any state fits within the pure data-oriented model used for scientific workflows. Finally, efficient implementation techniques have previously been developed by the functional programming community, which, as we demonstrate in this thesis, can be readily adapted for the purpose of workflow execution.

Our extensions to lambda calculus are purely convenience mechanisms. It is possible to define boolean values, cons cells, the `if` function, and even numbers in terms of pure lambda calculus, as we explain in Appendix A. Syntactic extensions like top-level function definitions and letrec expressions are also not fundamentally necessary, as anonymous functions (lambda abstractions) and other forms of expressions can be bound to identifiers by supplying them as parameters to other functions, and recursive functions can be defined using the Y combinator [161]. Our arguments about the simplicity and expressiveness of our model mostly pertain to the abstract model of lambda calculus. We provide extensions for reasons of practicality, in particular for ease of use and efficiency.

Lambda calculus can be considered an alternative to dataflow, the model upon which most existing workflow languages are based. Both models are abstract in nature and concern only program structure and evaluation semantics. We argue that lambda calculus is a more flexible approach for defining workflows, for the following reasons:

- Higher-order functions can be defined in lambda calculus, but are not normally supported in dataflow languages. Higher-order functions are a powerful means of abstraction which enable certain classes of control structures to be defined in user code, simplifying the structure of the workflow. With lambda calculus, standard higher-order functions such as `map`, `filter`, and `foldr` can easily be implemented in user code, whereas in dataflow languages they must be implemented as language extensions.

- In lambda calculus, iteration and recursion are naturally expressed in terms of function applications, which are evaluated according to the β-reduction rule. In contrast, dataflow languages require extensions such as dynamic graph expansion or tagged tokens in order to support iteration and recursion. Additionally, both types of control structures require loops in the dataflow graph, making graphical representation of the workflow awkward when multiple such structures are used in the one function.

- Creation and manipulation of data structures is readily supported in lambda calculus, even in the absence of native support from the language implementation. For example, the functions `cons`, `head`, and `tail` can all be defined as lambda abstractions without any built-in support for cons pairs. These can be used to build up lists and other types of data structures. Almost all functional languages support cons pairs natively, for convenience and efficiency.

- Language features are much more consistent between different functional languages based on lambda calculus, in comparison to dataflow languages. This is perhaps because in lambda calculus, tasks such as iteration, recursion, and list processing do not require special support from the language, whereas dataflow languages implement a diverse range of extensions to the basic dataflow model in order to support these features. Lambda calculus has a much greater mind-share in the functional programming community; a wide range of documentation and established conventions make languages based on the model easier for new users to adopt.

The following sections explain how concepts from workflow languages can be expressed in terms of ELC.

3.6.1 Tasks

A task in a workflow is an operation provided by some external entity, such as a web service. Tasks are conceptually equivalent to function calls — they take a series of inputs, perform some processing, and then produce a result. In existing workflow languages, each task is exposed as an atomic operation, and the service invocation logic is implemented within the workflow engine itself. In ELC, tasks implement encoding of arguments and decoding of results directly in user code, and use the networking functionality described in Section 3.5.8 to transfer data over the network.

For example, consider a task called `contains`, which takes two strings as input, and invokes a service over a simple RPC protocol in which a request is formed by supplying the operation name and arguments, separated by spaces, in a single line of text. To form a request, it is necessary to concatenate the operation name and a space, the first argument, another space, and finally the second argument. Assuming that the service is located on a machine called `host1` on port 2000, this task would be implemented as follows:

```
contains search within =
(connect "host1" 2000
  (++ "contains " (++ search (++ " " within))))
```

Tasks may be included within a workflow using the standard function call syntax, e.g.:

```
(contains "e" "welcome")
```

A more complex example might perform other processing on the input values, such as converting numbers to strings or serialising complex data structures into XML. Depending on the format of the response, it may need to be parsed, perhaps to extract a list of items or different components of a data structure. All of this logic may be expressed in ELC, using the built-in functions provided by the language. Another example is given in Section 3.7.2.

Task functions are equivalent to stub functions in traditional RPC systems. Although it is possible for users to write their own stub functions, it is more common to use a tool which generates these automatically based on an interface definition file, which contains information about the operations provided by the service such as their names and parameters. It is possible to write such tools to generate the stub functions such as the one shown above. ELC is designed as a generic language which is capable of supporting arbitrary service protocols and data formats, and thus automatic stub generation is outside its scope.

It is intended that other tools will be used in conjunction with ELC and NReduce which address higher-level issues such as service protocols and interface definitions. An example of this is our XQuery implementation, described in Chapter 5. It provides support for SOAP-based web services, and includes the ability to generate a set of stub functions based on the contents of a WSDL file, which defines the interface provided by a web service. Combined with a set of library routines which implement the HTTP protocol, the functions that our XQuery implementation generates ultimately use the `connect` function to send and receive the inputs and outputs of a service, in the same manner as the example above.

3.6.2 Conditional branching and iteration

Conditional branching can be achieved in ELC using the built-in `if` function, which accepts three parameters — the condition, the true branch, and the false branch. The first parameter is evaluated, and based on whether it returns true or false, either the second or third parameter is then evaluated. Although strict evaluation is used by default, the `if` function is treated specially, and evaluated lazily. This is to avoid both the true and false branches being evaluated, which would lead to infinite loops when the `if` function is used as part of a recursive function.

Iteration can be expressed in two different ways. The first is the traditional notion of iteration, where a body of code is executed repeatedly until some condition is met. As is usual in functional languages, each iteration of a loop is entered by making a tail recursive call. The body of a loop must therefore be a recursive function, in which the loop variable is one of the parameters. Each time a recursive call is made, an incremented version of the loop variable is passed as a new parameter. An example of this is as follows, where `from` is the loop variable:

```
sum from to total =
(if (> from to)
    total
    (sum (+ from 1) to (+ total from))
```

The same thing could also be achieved without using tail recursion, though this does not correspond directly to iteration, since the addition is performed after the recursive call:

```
sum from to =
(if (> from to)
    0
    (+ from (sum (+ from 1) to)))
```

Another common form of iteration is performing some processing on each value in a list. This can be achieved using `map`, which applies a function to each element in a list, and returns a new list containing the results of each function call. `map` is defined as follows:

```
map f lst =
(if lst
    (cons (f (head lst))
          (map f (tail lst)))
    nil)
```

When NReduce is run using the default strict evaluation mode, the `cons` function is strict in both arguments, and thus each will be considered a candidate for parallel evaluation. Each call to `map` sparks evaluation of the application of the function to that particular value, as well as the next recursive call to `map`. As a result, all of the list items can be processed in a data parallel fashion.

`map` can used to invoke a service for each item in a list of values as follows:

```
(map S lst)
```

where `S` is a function of one argument that invokes a service. For example, consider a service called `square` which multiplies its argument by itself. To obtain the squares of the numbers from 1 to 10, we could write:

```
(map square (range 1 10))
```

The function passed to `map` can also be defined in-place using a lambda abstraction, e.g. in the example shown below, which makes two service calls for each item, concatenating the results.

```
(map (\x.++ (svc1 x) (svc2 x)) lst)
```

Other higher-order functions common to most functional programming languages such as `filter` and `foldr` are supported as well. `filter` takes a list of items and a predicate function, and returns a new list containing only those items for which the predicate evaluates to true. `foldr` applies a binary operator to each item in the list as well as an accumulating parameter; an example of this is concatenating a list of strings:

```
(foldr ++ "\n" (cons "one" (cons "two" (cons "three" (cons "four" nil)))))
```

which is equivalent to:

```
(++ "one" (++ "two" (++ "three" (++ "four" "\n"))))
```

A major benefit of expressing iteration using recursive and higher-order functions is that these approaches are much more straightforward than those required to achieve iteration in dataflow graphs. The latter requires feedback loops which pass tokens from the output of one iteration of a loop back to the start [118]. Others have noted the awkwardness of this approach, since it requires explicit handling of the iteration in a non-trivial manner, and forces execution to be sequentialised [209]. Supporting operators like `map` and `filter` in workflow systems that do not support higher-order functions would require implementation within the workflow engine itself, significantly complicating the exploitation of parallelism, since these would have to be treated as special cases. The approach we have taken enables these functions to be easily defined in user code. It also enables us to rely on a single mechanism for triggering parallel evaluation of expressions, which is that when a call is made to a strict function of multiple arguments, all of the arguments are considered candidates for parallel evaluation.

3.6.3 Embedded scripting languages

Existing workflow languages usually provide the ability to include tasks written in one or more embedded scripting languages. This capability is provided so that programming logic that cannot be expressed directly in the workflow itself can still be specified. Since it is awkward or impossible to define computational logic in these workflow languages, users often need to switch to scripting languages instead.

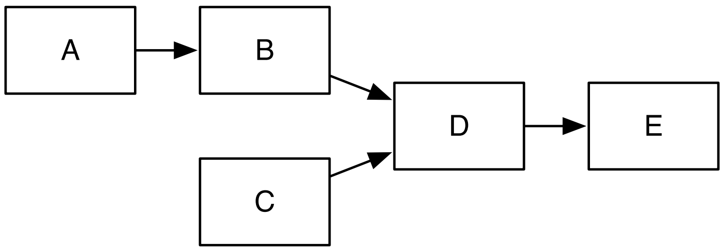


Figure 3.6: Example workflow graph

One of the key advantages of our programming model is that we use a *suitably general* language, which is capable of defining both the high-level *and* the low-level details of a workflow. As an abstract model, lambda calculus is independent of the granularity of operations, and provides a simple set of rules upon which many control flow constructs can be defined. In ELC, we provide basic operations for performing arithmetic and logical operations, as well as functions to create and access lists and other data structures built out of cons pairs. This enables logic that would otherwise need to be implemented in an embedded scripting language to be included directly within the workflow. For this reason, we do not provide direct support for embedded scripts written in other languages.

Since ELC is very generic in nature, it can also be used as a target language by compilers of other languages. Some languages provide built-in constructs for dealing with particular types of data structures, enabling certain types of processing to be expressed more concisely than would be possible when working directly in ELC. For example, XQuery contains many constructs for querying and manipulating data in XML format. In Chapter 5, we demonstrate how XQuery can be implemented in terms of ELC.

3.6.4 Representation

In existing workflow languages, programmers work with the conceptual model of a graph, in which they define a series of nodes and edges indicating tasks and data dependencies. Most often, the graph is edited using a graphical user interface, however some languages only offer a textual syntax. Even those workflow systems which do include graphical editors also include a textual syntax for storing a workflow graph when it is saved in a file or passed to the workflow execution engine.

The syntax that existing workflow languages use for text-based representation consists of a series of statements, each of which defines a node and an edge. The workflow from Figure 3.6 would be written something like this:

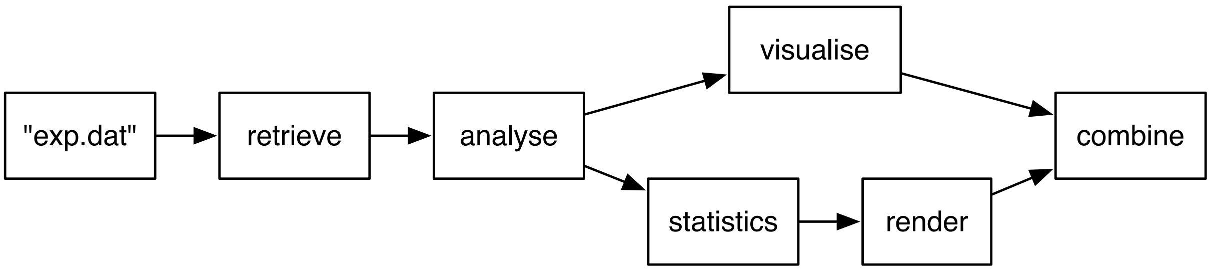


Figure 3.7: Example scientific analysis workflow

```
node A
node B
node C
node D
node E
edge A -> B
edge B -> D
edge C -> D
edge D -> E
```

The problem with this syntax is that it is extremely verbose — even without encoding the statements in XML, which many workflow languages insist on. Consider the alternative in ELC:

```
(E (D (B A) C))
```

This second case contains the same information as the first. All of the nodes can be written on a single line, and the data dependencies are implicit in the structure of the expression. Parsing this expression results in the same graph (or tree, in this case) as shown in Figure 3.6.

This expression syntax is sufficient for workflows in which each task's output is only ever passed as input to one other task. However, it is common for there to be tasks whose results are passed to two or more other tasks. In this case, a letrec binding can be used to establish a mapping between a symbol and an expression, so that symbol may be used multiple times to express the notion that multiple tasks share the same result value. For example, the graph shown in Figure 3.7 could be expressed as follows:

```
(letrec
   results = (analyse (retrieve "exp.dat"))
 in
   (combine (visualise results)
            (render (statistics results))))
```

Although letrec is a syntactic extension specific to ELC and other functional languages, the same thing can be achieved in pure lambda calculus using a lambda abstraction:

```
((\results.combine (visualise results)
                   (render (statistics results)))
 (analyse (retrieve "exp.dat")))
```

With the exception of lambda abstractions, the expression syntax we have described here is not inherently specific to lambda calculus; it could just as easily be used for dataflow languages as well. Unfortunately, many existing workflow systems are designed with the assumption that workflows will only ever be edited using the graphical interface, and therefore little attention has been paid to the usability of their text-based syntax. We suspect that one of the reasons for choosing the more verbose syntax is so that layout information for each node, such as its position on screen, can be saved in the workflow file. It would be relatively straightforward to extend existing workflow editors with the capability to import and export workflows in a more concise syntax such as the one we have described, though unfortunately none of the editors we are aware of provide such a feature.

The motivation behind the development of visual editing tools to is to make it easy for end-users with little programming knowledge to develop workflows [109]. Although these editors make it very easy to create trivial workflows involving a small number of tasks, developing more complex workflows involving control flow and data manipulation still requires a degree of programming skill, since workflow development is essentially a form of programming.

Even in purely graphical systems, developing complex workflows with existing tools requires programmers to be familiar with one or more scripting languages. Because of limitations in the feature sets of workflow languages, existing systems support embedded scripting languages in which tasks can be developed. This is necessary whenever something slightly outside the capability of the workflow language needs to be expressed, such as evaluating basic arithmetic expressions, or accessing parts of data structures. The surface-level appeal of a graphical editing environment is soon tarnished when a user discovers that in order to achieve everything they need, they are forced to implement part of their workflow in a traditional scripting language.

Learning syntax is the easy part of programming; understanding the semantics of a language and developing the appropriate problem-solving skills are the hard parts. For this reason, we see little benefit to visual editing environments, and believe that our efforts are best directed at providing a simple, concise, and easy-to-use text-based workflow language. As we have demonstrated above, lambda calculus provides a vastly superior syntax to the text-based serialisation formats of existing workflow languages. This approach, however, does not preclude the future development of third-party visual editors which output ELC code.

3.6.5 Abstract workflows

In lambda calculus, abstract workflows can be modeled as higher-order functions which take parameters specifying the concrete implementations of tasks. For example, the workflow from Figure 3.6 can be parameterised as follows:

```
(\A.\B.\C.\D.\E.E (D (B A) C))
```

This function can then be applied to a set of function parameters which implement tasks A through E. If none of the parameters contain any free variables, it is a concrete workflow which can be directly executed. Otherwise, it is a less abstract workflow that contains some implementation details, but is still parameterised. For example, the following abstract workflow specifies each task as a web service call to a specific operation, but is parameterised by the service URLs:

```
AWF =
(\urlA.\urlB.\urlC.\urlD.\urlE.  // Service URL parameters
  (\A.\B.\C.\D.\E.E (D (B A) C)) // Abstract workflow
    (wscall urlA ...)            // Task function for A (no parameters)
    (\x.wscall urlB ...)         // Task function for B (one parameter)
    (wscall urlC ...)            // Task function for C (no parameters)
    (\x.\y.wscall urlD ...)      // Task function for D (two parameters)
    (\x.wscall urlE ...))        // Task function for E (one parameter)
```

This abstract specification can then be instantiated into a concrete workflow by applying the function to a set of parameters specifying a specific set of services to be accessed:

```
(AWF "http://a.org/analyse"
     "http://b.org/filter"
     "http://c.org/query"
     "http://d.org/process"
     "http://e.org/genreport")
```

One application of the abstract workflow concept is to provide *quality of service* (QoS) mechanisms, whereby services are chosen at runtime based on certain requirements. For example, a user of the above workflow may want to specify that they want to use the cheapest service available that implements A, the fastest available versions of services B and C, and the most reliable versions of services D and E. Suppose that the workflow made use of a library of functions which could be used to locate services based on these criteria, perhaps by querying a service registry that maintained this information. Using these functions, the workflow could be expressed as follows:

```
(AWF (find_cheapest "a")
     (find_fastest "b")
     (find_fastest "c")
     (find_most_reliable "d")
     (find_most_reliable "e"))
```

Abstract workflows defined using these techniques are a flexible way of achieving reuse. A scientist may develop a workflow parameterised by service addresses, and then run it in their local environment by supplying the necessary set of URLs. The abstract workflow could subsequently be shared with other scientists, who may run it with the local versions of those services hosted at their own institution, or with a different set of input data. Such usage is equivalent to a script which allows its behaviour to be customised using a configuration file or command-line parameters, rather than relying solely on hard-coded information.

The ability to express abstract workflows in this manner does not require explicit support from the workflow engine. It comes about as a natural consequence of the inherent support for higher-order functions that is present in lambda calculus. It is one of the many examples of how a wide range of functionality can be built up from a minimal set of language constructs, instead of extending a monolithic infrastructure.

3.7 Example workflow: Image similarity search

Recall the case study introduced in Section 1.1, which concerns an image search function provided by a web site. This feature uses content-based image retrieval techniques [275] to allow users to search for images based on visual properties such as colour, text, and shape. The query is in the form of an image selected by the user, in response to which the site returns a list of other images that have a similar appearance to the query image. To enable results to be retrieved quickly, the search function relies on a pre-computed index which stores the similarity relationships between images. Since the indexing process is computationally expensive, it must be carried out by a large collection of machines working in parallel. In this section, we describe how a workflow to coordinate this process could be implemented using our programming model.

The back-end infrastructure provided by the site is based on a service oriented architecture, in which each machine in the server farm hosts a collection of services providing a range of functionality. Among these is a storage service, which provides access to a collection of files, and an image processing service, which performs various operations on images. The workflow in question needs to combine operations from these two services to retrieve the images, and compare every possible pairing of images to construct a graph containing the relationships between them.

The service operations used by the workflow are as follows:

- `get` — returns the image with the specified name
- `analyse` — performs feature extraction on an image, returning an image profile
- `compare` — compares two profiles, returning a similarity score in the range 0 – 1

For simplicity, we shall assume that the index to be produced by the workflow is represented in textual format, with each image in the database being listed alongside those images which are similar to it, as well as similarity scores indicating the strength of the relationships. A better approach would be to generate the index file in a format which is optimised for quick lookup operations. However, this would complicate the index generation part of the workflow, so we use a simple text-based representation here to keep the example code simple.

The basic structure of the workflow is as follows:

1. For each image, download the image, and then pass it to the analysis service to produce an image profile. The result of this step is a list of (name, profile) records
2. For each image:
   (a) Compare the profile of this image against the profile of every other image, producing a list of tuples, each of which contains the name of the other image and its degree of similarity to this image
   (b) Filter the list of tuples to remove those with a similarity score less than 0.9
   (c) Sort the list of tuples according to their similarity score, so that those with the highest scores appear first
   (d) Construct a string containing the name of the image, followed by the names and scores of all other similar images
3. Concatenate the strings produced by all loop iterations carried out in step 2

3.7.1 Workflow implementation

Figure 3.8 shows the implementation of the workflow, which is defined as a function called `build-index` that takes a list of image names as parameter.

Step 1 iterates over this list. In each iteration, it invokes the `get` operation to retrieve an image, and passes this image to the `analyse` operation. The resulting image profile contains information about the image, which will later be compared with other image profiles. The image name and profile are passed to the `make-record` function, which places them both in a tuple, which we refer to here as being of type `record`. The result of step 1 is a list of (name, profile) records.

Looping is achieved using `map`, a higher-order function which takes two parameters — a function and a list — and applies that function to every element in the list. The result of `map` is another list containing the results of each function application. In this workflow, the first parameter to `map` is always a lambda abstraction (anonymous in-line function definition) containing the loop body. This technique is used in step 1 to loop over the list of image names, and in steps 2 and 2a to loop over the list of (name, profile) records.

Step 2a compares the current image profile against every other image profile. The loop body invokes the `compare` operation, passing in as parameters the profiles extracted from the record tuples of both images. Once the comparison is complete, `compare` returns a numeric score in the range 0 – 1, indicating how similar the images are. The loop body combines the name of the other image and the similarity score into a tuple of type `compres`

```
build-index image-names =
(letrec
  records = (map (\name.make-record name                        // Step 1
                                   (analyse (get name)))
                 image-names)
  results =                                                     // Step 2
    (map (\this.                                                // (loop)
           (letrec
              comparisons =                                     // Step 2a
                (map (\other.
                       (make-compres
                         (record-name other)
                         (compare (record-profile this)
                                  (record-profile other))))
                     records)
              best-matches =                                    // Step 2b
                (filter (\a.>= (compres-score a) 0.9)
                        comparisons)
              sorted-matches =                                  // Step 2c
                (mergesort (\a.\b.- (compres-score b)
                                    (compres-score a))
                      best-matches)
            in
              (++ (record-name this) (++ "\n"                   // Step 2d
              (foldr ++ "" (map compres-to-string
                                sorted-matches))))))
         records)
in
  (foldr ++ "" results))                                        // Step 3

compres-to-string compres =
(++ "  " (++ (compres-name compres)
(++ ": " (++ (ntos (compres-score compres)) "\n"))))

make-record name profile = (cons name profile)
record-name record       = (head record)
record-profile record    = (tail record)

make-compres name score  = (cons name score)
compres-name compres     = (head compres)
compres-score compres    = (tail compres)
```

Figure 3.8: Workflow for constructing an image similarity index

```
mergesort cmp m =
(if (<= (len m) 1) m                                  // empty/single; return
(letrec middle = (floor (/ (len m) 2))                // middle = len(m)/2
        left = (mergesort cmp (prefix middle m))      // sort left half
        right = (mergesort cmp (skip middle m))       // sort right half
in      (merge cmp left right)))                      // merge sorted halves

merge cmp xs ys =
(if (not xs) ys                                       // xs empty; return ys
(if (not ys) xs                                       // ys empty; return xs
(if (< (cmp (head xs) (head ys)) 0)                   // xs[0] < ys[0] ?
    (cons (head xs) (merge cmp (tail xs) ys))         // add xs[0] to list
    (cons (head ys) (merge cmp xs (tail ys))))))      // add ys[0] to list
```

Figure 3.9: Merge sort

(comparison result), which is later used during sorting and filtering. The reason why we return a list of tuples instead of just a list of scores is that when sorting the list, we need to maintain the association between image names and scores.

Constructor and accessor functions for `record` and `compres` tuples are defined at the bottom of Figure 3.8.

Step 2b constructs a new list containing only those comparison results whose similarity score is at or above a threshold of 0.9. This step uses `filter`, which is similar to `map`, except that instead of returning the results of applying the specified function, it returns only those items from the original list for which the function application evaluates to true. In this case, the function parameter is a lambda abstraction which tests whether the image score is greater than or equal to the threshold value.

Step 2c sorts the filtered list of comparison results according to similarity score, with those having the largest scores appearing first in the resulting list. The first parameter to `sort` is an ordering function, which takes two arguments corresponding to items from the list, and returns either a positive, zero, or negative value if the first argument is greater than, equal to, or less than the second argument, respectively. In this ordering function, the score associated with the first comparison result is subtracted from the score associated with the second. If the first score is greater than the second, the result will be negative, causing the first tuple to appear earlier in the sorted list. If the first score is less than the second, the result will be positive, causing the first tuple to appear later in the sorted list.

The sorting algorithm is also implemented in ELC, as shown in Figure 3.9. Due to the fact that side effects are prohibited, neither the `mergesort` or `merge` functions can modify their arguments; instead, they return a newly constructed list containing the items in the appropriate order. These functions rely on recursion not just for sorting sub-lists, but also for iterating over the sorted lists when merging them together.

Step 2d produces a string containing the name of the current image, and the names and

```
compare profile1 profile2 =
(ston (http::post "http://image-processing.local/apis/compare"
                  (++ (flatten-numbers profile1)
                  (++ "\n"
                  (flatten-numbers profile2)))))

flatten-numbers list =
(foldr ++ "" (map (\num.++ (ntos num) " ") list))
```

Figure 3.10: Implementation of the `compare` task

scores from the sorted list of similar images. The `++` function is used to concatenate the name of the current image, a newline character, and the strings produced for each item in the sorted list. These strings are produced using the `compres-to-string` function, which returns a string containing the name of the other image and its similarity score. The strings returned by `compres-to-string` are combined by a call to `foldr`, which reduces the list of strings into a single string using the concatenation operator, in the manner demonstrated in Section 3.6.2. This same technique is used in step 3 to combine the strings from all the outer loop iterations.

3.7.2 Task implementation

There are many different protocols that could potentially be used by the services. In this example, we shall consider the case of a simple REST service, which operates by having the client make a HTTP POST request to the following URL:

`http://image-processing.local/apis/<opname>`

Let us consider the `compare` operation, which takes two image profiles as input. Other than header fields, there is no specific syntax imposed by HTTP on the body of request or response messages, so it is up to the client and server to agree on a format. Suppose that each image profile is represented within the workflow as a list of floating point numbers, and that the server accepts input in the following format:

- A space-separated list of numbers representing the first profile
- A newline character
- A space-separated list of numbers representing the second profile

This scheme implies that there must be three main parts to the task implementation: constructing a string containing the values of a profile object, concatenating the serialised representations of the two profile objects together, and passing the request to the server.

Figure 3.10 shows the code for the `compare` function. This first constructs a request body containing string representations of the two image profiles, then passes this to the `http::post` library function, which builds a request by prepending the appropriate HTTP headers to the body, then uses the built-in `connect` function to establish a connection to the server and pass it the request. It then parses the response to check that the operation completed successfully, and extracts the result of the operation from the service. In this case, the result of the `compare` operation is a floating point number, so the response is passed to the `ston` function, which converts a string to a number.

The `flatten-numbers` function takes a list of numbers representing an image profile object, and concatenates them together into a string, separated by spaces. This is achieved by first using `map` to produce a list in which each item is the string value of the number, followed by a space. This list is passed to `foldr`, which concatenates all of the strings together in the same manner as described previously.

This example is just one of the ways in which the task could implemented. Other service protocols or input formats would require a different implementation. For example, if the `compare` operation were provided by a SOAP-based web service, the request would need to be formed as an XML document containing a SOAP envelope and elements for the operation name and parameters. An advantage of SOAP-based web services is that they provide a formal description of the operations and message formats provided by the service in the form of a WSDL file, enabling the task functions to produced automatically by a code generator. We use this technique in our XQuery implementation, allowing programmers to call task functions without having to implement them manually, like in this example.

3.8 Support for higher-level languages

ELC is an extremely simple yet general notation for defining workflows, and is capable of expressing any possible computation. However, since it only provides a few basic primitives, it requires programmers to specify their logic in great detail. In some cases, it is preferable to instead use a higher-level language containing primitives that enable certain aspects of a program to be expressed more concisely.

One of our reasons for choosing such a general model is that it can be used as an intermediate form by compilers of other languages. Just as some compilers for functional languages use C as a target language to separate out architecture-dependent concerns [163, 314], ELC can be used as a target language to separate out parallelism and distribution concerns from high-level language issues. In this sense, ELC can be considered a generic model capable of supporting many different data-oriented workflow languages.

To demonstrate this style of usage, we developed an implementation of XQuery [43], a functional language designed for manipulating XML data. Our implementation, discussed in Chapter 5, extends XQuery with support for web services, enabling it to be used as a workflow language. To achieve this, we developed a compiler which translates XQuery source code into ELC, and a library of support functions which are used by the generated code.

The most important advantage of XQuery is its native support for XML processing. In the context of web service-based workflows, this makes it useful for performing intermediate manipulation of data exchanged between services, the importance of which was discussed in Section 3.3.4. The language constructs provided by XQuery enable XML manipulation to be expressed much more concisely than is possible when working directly with ELC. Built-in support for parsing and serialisation is also important for exchanging data with services and producing the final output of the workflow.

Stub functions, such as the example given in Section 3.7.2, are necessary to enable services to be accessed within a workflow, but are awkward to construct manually. SOAP-based web services provide formal interface definitions in the form of WSDL files [66], from which it is possible to automatically generate stub functions. Our XQuery implementation supports the `import service` statement [243], with which the programmer can specify the URL of a WSDL file, and the compiler will automatically generate the stubs for all of the operations that service provides. This relieves the programmer of the responsibility of manually implementing these functions themselves, greatly simplifying the task of using web services.

Our approach of providing a highly generic programming model means that the fundamental abstractions it provides can be used by implementations of a wide range of workflow languages. Automatic detection of parallelism, invocation of multiple concurrent service calls, and choreography of services across multiple hosts are addressed entirely within the virtual machine, and do not need to be addressed by higher-level language implementations. Compilers for these can focus on the semantics of the language in question, without having to be concerned with how execution is achieved.

3.9 Summary

In this chapter, we have motivated and described a programming model called ELC, which is based on lambda calculus. This programming model is designed for data-oriented workflows, and leverages existing concepts from functional programming languages. In comparison with the restrictive models of existing workflow languages, ELC provides a small yet flexible set of language constructs which can be used to express complex workflows involving application logic and data manipulation. Network connections are exposed to the language in a generic manner, which enables many different protocols to be supported on top of the primitives provided, instead of having these mechanisms hard-coded into the workflow engine.

The model we have presented is very similar to other functional languages; the main aspects that require consideration for workflows are to do with implementation, in particular support for automatic parallelisation and concurrent handling of network connections, which are discussed in the next chapter. The purely data-oriented semantics of functional languages and their elegant theoretical models, based on lambda calculus, are in our opinion highly appropriate for workflows. We argue that by adopting functional programming ideas, workflow systems can be built on a stronger theoretical basis and with greater flexibility than current approaches permit.

ELC is intended both as a language in which workflows can be written directly by a programmer, and as an intermediate language which can be targeted by compilers of other, higher-level workflow languages. We have demonstrated this idea by introducing our implementation of XQuery, described in Chapter 5, which includes a compiler that generates ELC code. This approach to workflow language development enables a single execution mechanism to handle operational concerns such as parallelism and network access, separating these issues from the high-level logic. As such, the NReduce virtual machine, described in the next chapter, is theoretically capable of supporting many different workflow languages.

In addition to language features, we have also addressed runtime issues regarding how data is transferred between machines participating in the execution of a workflow. The most common approach, orchestration, requires data to be transferred between the client and services on every call. An alternative approach, choreography, supports direct interaction between the machines, allowing the total amount of data transferred to be reduced by sending data directly between producers and consumers. We have proposed an approach which requires no explicit involvement of the services themselves, enabling them to use the standard request/response model, by co-hosting nodes of a distributed virtual machine with the services, with the virtual machine taking care of all data transfers. The details of how this choreography occurs is an implementation consideration of the virtual machine, and is entirely abstracted away from the programming model.

In the following chapters, we describe our implementation of the ideas introduced in this chapter. It is worth noting that ours is only one possible implementation, and there are many other ways in which the concepts discussed here could potentially be realised. It is hoped that the ideas in this chapter will serve useful to those designing other workflow systems in the future.

Chapter 4

The NReduce Virtual Machine

A central aspect of a workflow system is its *execution engine*. This is a piece of software which takes workflow definitions as input, and executes them by invoking tasks, transferring data, and performing intermediate computation on the inputs and outputs of tasks. Most workflow engines are implemented in a monolithic fashion, where both the execution logic and high-level functionality such as service access mechanisms are implemented directly within the same piece of software. The languages supported by such systems tend to be very simple and restrictive, generally supporting only high-level operations, and limited or no data manipulation. The often high per-operation overhead and lack of basic data types and operators makes it impractical to implement application logic or support functionality directly in these languages.

The approach proposed in this thesis is to instead make the workflow engine deal with only low-level concerns, and construct it in a manner similar to a regular functional programming language implementation. By supporting a small set of basic primitives, and executing code with a high degree of efficiency, it is possible to implement high-level functionality in terms of the language supported by the engine. This separation of concerns keeps the execution engine simple by only requiring it to implement a small set of features. It also enables a greater degree of customisation by developers, who may implement a wide range of functionality on top of this platform, without having to modify the engine itself.

Another benefit of separating the execution mechanisms and high-level functionality is that multiple workflow languages can be supported on top of the one execution engine. This can be achieved by having a separate compiler for each language, each of which outputs code in the input language understood by the engine. Provided that each supported workflow language fits within the general execution model of the engine, the translation is relatively straightforward, as it simply needs to express the constructs of the workflow language in terms of the appropriate low-level operations.

We strongly believe that functional programming offers an ideal model for data-oriented workflows. However, none of the existing workflow systems described in Section 2.3.7 can properly be considered implementations of functional programming languages. On the other hand, existing work on functional programming assumes that all computation is internal to the program — in contrast to workflows, which delegate most of their computation to external tasks. In this chapter, we describe how we have taken well-established implementation techniques from the functional programming community, and adapted them in a manner suitable for workflow execution.

NReduce is a virtual machine which implements the ELC programming language described in Section 3.5. It is designed to be used as a workflow engine, demonstrating the way in which functional programming can be used as a model for expressing and evaluating workflows. Distinguishing features of NReduce include automatic detection of parallelism, data-oriented abstraction of network connections, lightweight threading support for asynchronous I/O, a compact array-like representation of cons lists, efficient execution via native code generation, and distributed execution across multiple nodes to support workflow choreography. NReduce can be used to run workflows either written directly in ELC, or written in other languages which are then compiled to ELC — an example of which is our XQuery implementation described in Chapter 5.

Many of the implementation techniques used in NReduce are based on established principles from past work in the functional programming community, discussed in Section 2.4.5. In particular, the abstract machine design, graph representation, and parallel execution mechanisms are derived from work by Peyton Jones [161], Augustsson and Johnsson [23, 24], and Trinder et. al. [300]. The contribution of this chapter is to demonstrate how these techniques can be applied to the problem of workflow execution, and to explain the design and implementation choices we have made based on our particular requirements. Several aspects of our implementation are also unique, including the way in which network connections are supported, and the relationship between these mechanisms and parallel evaluation.

4.1 Goals

Our goals when developing ELC and NReduce were as follows:

- **Support for data-oriented workflows.** As described in Section 2.3.1, data-oriented workflows utilise side effect free remote tasks, and coordinate them using explicitly defined data dependencies. The functional programming model is well-suited to these types of workflows, which is why we have chosen it as the programming model for our virtual machine. In language implementation terms, each remote task is invoked by calling a function which uses the networking features of the runtime system to communicate with the service, passing it the appropriate input data and retrieving the result. These service access functions provide the programmer with the ability to call remote tasks in the same manner as normal functions.

- **Language simplicity.** NReduce is intended to be a platform on which multiple high-level workflow languages can be executed, via compilation to an intermediate language, ELC. This means that all of the constructs of these source languages will be handled by their respective compilers, which are responsible for using whatever features are provided by the intermediate language to implement them. Because of this, and to avoid unnecessary implementation complexity, it was decided that the virtual machine should only support a very simple programming language. Supporting an existing fully-featured functional language would have involved much more development work, for little benefit.

- **Automatic parallelisation.** Most data-oriented workflow engines handle parallelisation automatically, which simplifies the job of the programmer by not requiring them to manually specify which parts of the workflow should run in parallel with others. This is feasible because the external tasks invoked by workflows are coarse-grained, so a moderate degree of per-operation overhead can be incurred while still allowing the performance benefits of parallelism to be obtained. Most existing parallel functional languages instead require parallelism to be specified manually, because they are designed for compute-intensive programs containing a large number of fine-grained operations for which the overheads of automatically managing parallelism for every expression would be too expensive. Since our engine is designed for workflows containing only small to moderate amounts of internal computation, we decided to detect and manage parallelism automatically.

- **Efficient execution of local computation.** Although the workflows we are targeting delegate a lot of their computation to external tasks, we still want to permit some of the application logic to be contained within the workflow itself. This means that it must be possible to express computation and data manipulation as locally executed functions. Also, the fact that the virtual machine only provides a small set of primitives means that support functionality such as data encoding and parsing of service responses must be handled in user code. The speed at which this code is executed should therefore be reasonably fast — at least competitive with mainstream interpreted scripting languages. However, it does not need to be competitive with compiled languages such as C and Java, since we work on the assumption that all compute-intensive logic will be implemented in external services.

- **Appropriate abstractions for network connections.** Parallelism in functional languages relies on the lack of side effects, enabling multiple independent expressions to be evaluated at the same time. In order to support parallel invocation of services, we therefore need to make it possible to invoke services from within pure, side effect free expressions. Existing functional languages take the overly restrictive view that all I/O involves side effects, forcing the programmer to specify a strict sequential ordering of I/O operations. This fundamentally conflicts with our desire to support invocation of multiple service calls in parallel, so we came up with the alternative approach described in Section 3.5.8. This approach involves exposing each connection as a side effect free function which maps from input data to output data.

4.2 Execution environment

The basic model of execution used by NReduce is *graph reduction*, introduced in Section 2.4.5. Like many other functional language implementations, NReduce does not rely on explicit traversal and manipulation of the graph, since doing so involves a lot of individual steps and temporary memory allocation for each function call. Instead, it uses an abstract machine to execute compiled bytecode which is generated from the original source. However, the basic notion of representing the program state as a graph and performing a series of transformations on it remains an important aspect of the execution model.

NReduce consists of both compile time and runtime components. The compiler translates ELC code into a set of abstract machine instructions represented in bytecode. This bytecode is then executed by the runtime part of the virtual machine, using an execution engine that implements this instruction set. Several other runtime support components are also provided, including built-in functions, access to network connections, memory allocation, and garbage collection. These are invoked either as a result of the execution engine encountering certain instructions, or in response to external events such as data arriving over a network connection. In the latter case, an interrupt-like mechanism is used to suspend execution of the program while the appropriate support routine is called.

The execution engine is implemented in two forms, which were constructed in succession. The first is an interpreter, which consists of a C implementation of each of the opcodes, as well as processing loop to dispatch to the appropriate opcode handler as bytecode is executed. The second is a native code generator, which produces machine-executable code by translating each bytecode instruction into a sequence of native CPU instructions. This provides significantly better execution performance by avoiding many of the runtime tests and instruction pointer updates required by the interpreter. Both the native code engine and the interpreter make use of the same support routines and in-memory graph representation.

4.2.1 Frames

The basic unit of execution context within the virtual machine is a *frame*. These frames correspond to function calls, and play much the same role as stack frames (function activation records) in imperative languages. A frame contains a *data area*, which is used to store parameters that are passed to the function, as well as temporary variables used during evaluation. It also has an *instruction pointer* associated with it, which refers to the address of the current instruction being executed, or in the case of frames that have yet to begin execution, the first instruction.

Frames exist within the graph as a particular type of object, along with data values, as shown in Figure 4.1. Pointers between frames correspond to calling relationships, and pointers from frames to data values represent parameters and variables used within a function. A frame may represent an unevaluated expression, or *closure*, which can be stored in memory and possibly begin evaluation at a later point in time if its result

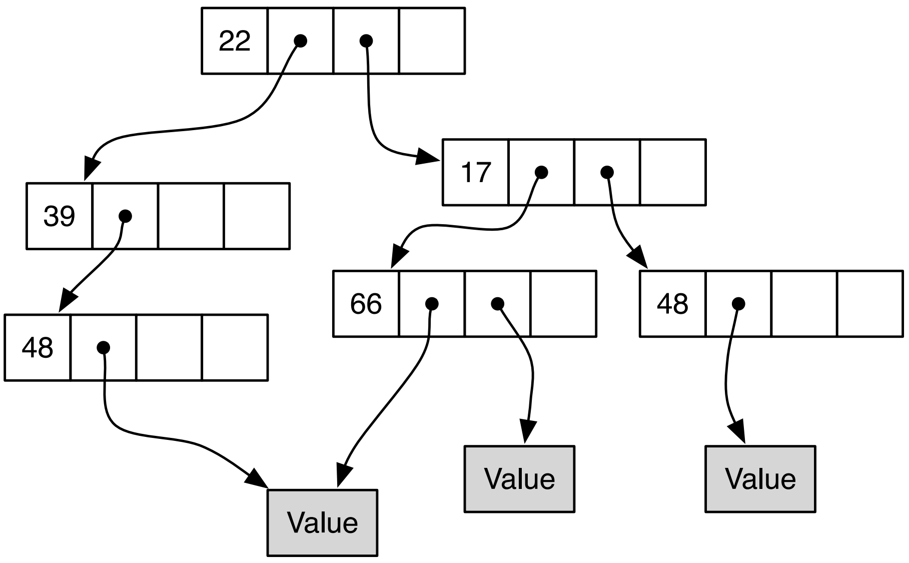


Figure 4.1: A graph containing frames and data values. The left-most field in each frame contains the instruction pointer.

value is needed. Alternatively, a frame may be *runnable*, in which case is a candidate for execution by the processor, or *blocked*, meaning that it is waiting on the result of another frame.

Because frames are organised in a graph rather than a stack, it is possible for more than one frame at a time to be eligible for execution. This opens up the possibility of parallelism, because in the distributed execution mode, different machines may execute different frames concurrently. Additionally, when a frame becomes blocked on an I/O operation, other eligible frames may be executed by the processor while the I/O operation is in progress. This is important for enabling multiple service calls to run in parallel. It also provides a form of latency hiding, which allows better utilisation of a processor in cases where a program is performing a significant amount of network interaction. This is particularly useful for workflows, which invoke operations on remote services.

At runtime, the execution engine maintains a reference to the current frame that is being executed. This is used to determine the area of memory in which parameters and variables reside. While the interpreter stores this in a local variable, the native code engine uses a register, allowing the relative memory locations to be looked up using the processor's built-in base+offset addressing mechanism. This register is used in the same manner as the base pointer in imperative languages. The current frame pointer changes whenever a function call or return occurs, or if a call to a blocking function is made and there is another runnable frame eligible for execution.

4.2.2 Abstract machine

At runtime, execution is performed by an implementation of an abstract machine which is based on an instruction set specifically tailored towards graph reduction. The bytecode generated during compilation of ELC code resembles a high-level assembly language, with various opcodes defined for data manipulation, control flow, and primitive operations such as arithmetic. Memory accesses occur relative to the current frame's data area, where all parameters and temporary values are stored. Opcodes are also provided to invoke function calls and evaluate closures.

The instruction set follows the same basic design approach as the G-machine [157], which introduced the idea of compiling functional expressions into instruction sequences that would produce the same result as a naïve graph reduction algorithm, but more efficiently. The reason for this efficiency is that an algorithm which performs graph reduction via explicit traversal of the graph has to perform many conditional tests at runtime, such as the type of nodes and the number of arguments supplied to a function. When using compilation however, it is possible to statically determine what the result of many of these conditional tests will be, enabling them to be skipped at runtime. The generated bytecode resembles what would be produced by a compiler of an imperative language, but handles features common to functional languages, such as closures and lazy evaluation.

The bytecode may either be interpreted, or compiled to native code. In our prototype implementation, we have not gone to the extent of fully optimising the native code generator. Although it saves the cost of opcode checking and dispatching, and permits more optimised implementations of many of the opcodes, it does not take advantage of registers to the extent that a regular compiler would. Each native instruction sequence generated for a given bytecode instruction updates the frame's data area according to the semantics of that instruction, which involves more memory access than optimised code that avoids saving and loading registers wherever possible. However, the native code implementation still provides performance benefits over a purely interpreted approach.

From the perspective of the compiler, the instruction set provided by the abstract machine conceptually treats the data area within a frame as a stack. Arguments to a function are placed at the bottom of the stack, and slots above these are used for temporary variables needed for expression evaluation. Expressions are compiled by generating instructions to push values onto the stack, and make calls to built-in operators that access their arguments by popping values from the stack and pushing the result.

For reasons of execution efficiency, the bytecode generator makes an additional pass over the compiled code, computing the size of the stack for each instruction and recording this value in the bytecode. This additional information effectively makes the executed bytecode equivalent to three address code, avoiding the need for the stack size to be dynamically updated at runtime. During execution, stack positions referenced within the bytecode instructions are treated as offsets in memory relative to the start of the current frame.

Most of the time, execution proceeds by executing instructions one-by-one. The majority of the opcodes simply perform some sort of modification to the current frame's data area, such as pushing values onto the stack, or invoking built-in functions. Standard control flow opcodes such as conditional and unconditional jumps are also supported. The full set of opcodes is listed in Table 4.1.

Opcode	**Description**
GLOBSTART	Marker for start of function
SPARK n	If node at stack position n is a frame, mark it as sparked, making it a candidate for parallel evaluation
RETURN	Return from the current function
DO	Inspect the value at the top of the stack. If it's a function, take parameters from below it on the stack and invoke it. If insufficient parameters are available, put them into a CAP (curried application) cell, representing an incomplete call to that function.
JFUN f	Jump to start of function f, without changing current frame. Used for tail calls.
JFALSE n	If value at top of stack is nil, jump ahead by n instructions
JCMP n f	Compare the top two values on the stack using numeric comparison operator f, and jump ahead by n instructions if the comparison succeeds
JEQ n v	Check if the value at the top of the stack is equal to v, and if so, jump ahead n instructions
JUMP n	Unconditionally jump ahead by n instructions
PUSH n	Push value at position n on top of stack
PUSHNIL	Push a nil value on to the stack
PUSHNUMBER n	Push the number n on to the stack
PUSHSTRING n	Push the string constant stored at position n in the string table on to the stack
POP n	Remove the top n values from the stack (optimised away at compile time, since stack pointer not used during execution)
UPDATE n	Remove value from top of stack and make graph node at position n an indirection node which points to it
ALLOC n	Allocate n empty cells which will later be updated with values, and place them on the top of the stack
SQUEEZE x y	Shift the top x values downwards in the stack, removing the y values below them
MKCAP f n	Place the top n values from the stack into a CAP cell, representing an incomplete call to the function f
MKFRAME f n	Place the top n values from the stack into a FRAME cell, representing a call to the function f
BIF f	Invoke built-in function f
ERROR	Abort execution, report a previously-recorded error message
EVAL n	Check stack position n, and if it refers to an unevaluated frame, begin evaluating it and block the current frame
CALL f n	Make a call to function f, passing the top n stack values as parameters. Equivalent to MKFRAME+EVAL.
CONSN n	Create a cons list containing the top n values from the stack
ITEMN n	Obtain the nth item of the cons list at the top of the stack

Table 4.1: Opcodes

```
Stack Addr Opcode       Arg0 Arg1
size

1      7   GLOBSTART    70   1    // Start of function #70
1      8   EVAL         0         // Evaluate parameter n
1      9   PUSH         0         // Push n on to stack
2     10   JEQ          10   1    // Jump to 20 if n == 1
1     11   PUSHNUMBER   1         // Push 1 on to stack
2     12   PUSH         0         // Push n on to stack
3     13   BIF          -         // Subtract 1 from n
2     14   MKFRAME      70   1    // New frame; recursive call with n-1
2     15   SPARK        1         // Spark stack position 1
2     16   PUSH         0         // Push n
3     17   EVAL         1         // Evaluate recursive call
3     18   BIF          *         // Multiply result by n
2     19   RETURN       0         // Return result n*fac(n-1)
1     20   PUSHNUMBER   1         // Push 1 on to stack
2     21   RETURN       0         // Return 1
```

Figure 4.2: Bytecode example

An example of the generated bytecode is shown in Figure 4.2. This represents the code produced for the following function definition:

```
fac n = (if (== n 1)
             1
             (* n (fac (- n 1))))
```

4.2.3 Data types and graph representation

ELC is a dynamically typed language, and supports only a small, fixed number of data types. The reason for this choice was to keep the programming model and implementation simple, by avoiding the need for a static type system.

The basic data types supported by the language are numbers, cons cells, functions, and the special value *nil*. Numbers are always stored in 64-bit double precision floating point format, so that it is not necessary to implement different sets of operations for different numeric types, or provide ways to convert between them. Strings are represented as cons lists in which each element is a number corresponding to a character code.

Although there is no built-in support for record types, it is possible to use cons lists to represent data structures by having each element of the list correspond to a different field. Section 4.6 discusses optimisations to the list representation that make this almost as fast as regular objects.

Each type of value, other than numbers, corresponds to a heap-allocated object stored in the graph, referred to as a *cell*. References to values from either within the graph or from

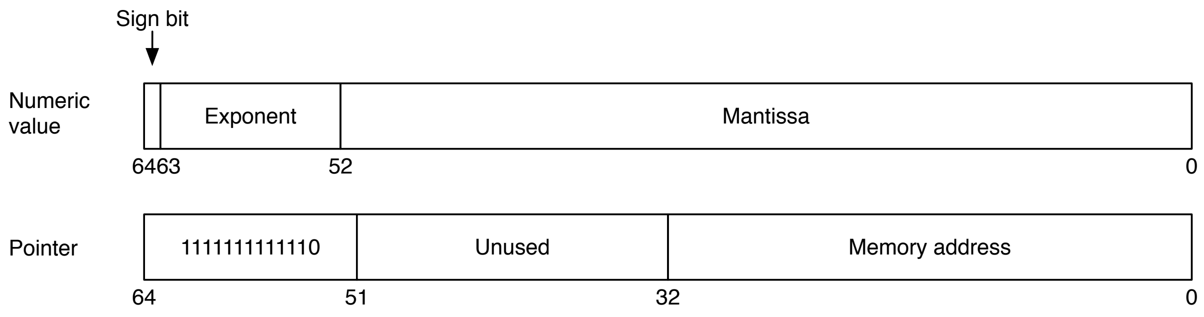


Figure 4.3: Encodings used for numeric values and pointers

the data area of frames are stored in a 64-bit format, which can either hold a number or a pointer. This means that numeric values do not incur a memory allocation cost. The 64-bit pointer/value representation is discussed further in Section 4.2.4.

Functions are only exposed as a single type to the programmer, in the sense that they all share the property that they are objects that can be called. However, there are three separate internal representations used, depending on the type of a function. The first type is *built-in* functions, corresponding to the functions implemented natively within the virtual machine, such as arithmetic operators. The second type is *supercombinators*, which are functions defined in user code. The third type is *CAP* (*curried application*) cells, which represent a call to either a built-in function or supercombinator in which only some of the arguments have been supplied, and appear to the programmer as functions which take the remaining arguments.

4.2.4 Pointers and numeric values

Memory allocation and garbage collection tend to have a significant influence on the performance of high-level languages. It is good to avoid allocation whenever possible, such as when dealing with primitive values. Some programming language implementations permit certain types of primitive values such as numbers to be stored in *unboxed* format, which means that they are passed around directly, instead of being stored as a heap object. Given that numbers are used very frequently by many programs, this can significantly reduce the amount of memory allocation and garbage collection that needs to be performed. It is for this reason that we chose to represent all numbers as unboxed values.

Two other concerns influenced the way in which we represent numbers. One is that supporting different numeric types such as floats and integers in a dynamically typed language incurs a runtime performance cost associated with type checking and conversion when performing arithmetic operations. Another is the need to distinguish between pointers and numbers, to avoid situations in which a numeric value could be interpreted as a memory address or vice-versa. For these reasons, we chose to represent all numbers as 64-bit doubles, and encode 32-bit pointers within the value space of NaN (not a number) values.

The IEEE 754 floating point format used by modern processors interprets all values with a non-zero mantissa and all exponent bits set to one as a NaN value. Because this encompasses a large set of values, it is possible to use the non-exponent bits to store other information, known as the *payload*. We take advantage of these extra bits to store the memory address that a pointer refers to. This scheme makes it easy to distinguish between a pointer and a number. We use so-called *signaling* NaNs for representing pointers, which causes the processor to raise a floating point exception if an attempt is made to pass a pointer to an arithmetic operation. Rather than incurring the cost of a type check before each operation, we rely on the processor's exception handling mechanism, which causes the operating system to send a signal to the process when an invalid argument is supplied to a floating point operation.

Figure 4.3 shows the ways in which numbers and pointers are represented. Numeric values use the standard IEEE 754 representation, which includes a sign bit, exponent, and mantissa. For pointers, the top 12 bits are set to one, bit 51 is set to zero (indicating a signaling NaN), and the lowest 32 bits store the memory address. The remaining 19 bits are only used in the case of array references, discussed in Section 4.6.2.

It should be noted, of course, that this representation assumes a 32-bit memory model. At the time we made this decision, 64-bit processors were far less common than they are today. However, the types of workflows we are targeting require well under 4 Gb of memory, so we do not anticipate this being a problem. If 64-bit support were needed, unboxed values would likely require type inference logic to be added to the compiler so that the representation to be used for a given value could be determined statically.

4.3 Parallelism

An important feature of workflow engines is the ability to have multiple external tasks running in parallel. This is because tasks are often executed remotely on different computers, and thus it is desirable to be able to have multiple computers running tasks at the same time. Achieving this requires the execution engine to initiate remote task execution *asynchronously*, so that it does not cause the whole workflow to block while waiting for a task to complete. Instead, parts of the workflow which do not depend on a given task should be able to continue executing while the task is in progress. This contrasts with the blocking operation of remote procedure calls exposed as regular functions in imperative languages.

Our approach avoids client-side blocking while still permitting remote services services to be accessed in the same manner as a local function call. The graph can contain multiple frames that are *runnable* (eligible for execution) at a given point in time, and the virtual machine is able to context switch between these when one of them blocks. This context switching is of a similar nature to that which occurs in a multi-tasking operating system when one process blocks on an I/O operation; in order to keep the CPU busy, another process is made the current one and continues execution. Our context switches happen entirely in user space, avoiding the costs associated with user/kernel mode switches and kernel-level threads.

We refer to the notion of having multiple remote operations in progress at the same time as *external parallelism*, which we distinguish from *internal parallelism*. External parallelism involves machines interacting solely via request/response messages to invoke service operations, and does not require the machines which are hosting services to be involved with the details of local code execution/graph reduction. Internal parallelism requires much more extensive support from the runtime system, which in the context of graph reduction involves each processor having access to shared data structures (the graph and spark pool) and synchronising with other processors when two or more need to access the same portion of the graph. Although our virtual machine supports both types of parallelism when running in distributed mode, our focus is on external parallelism, which is of most relevance to workflows. Internal parallelism has already been studied extensively by others, as discussed in Section 2.4.5.1, and is only supported by our virtual machine in order to facilitate workflow choreography.

4.3.1 Automatic detection vs. manual specification

Two basic approaches to identifying parallelism are used by the existing work described in Chapter 2. One is to inspect the data dependencies within the program and use this information to extract the maximum amount of parallelism possible. This is appropriate for programs involving coarse-grained operations, in which the average amount of computation performed by each operation is large in comparison to the overhead involved with managing the parallelism. The other approach is to not to detect parallelism automatically at all, but instead rely on programmer-supplied annotations that specify which expressions should be evaluated in parallel with others. This latter approach is more appropriate for programs involving large amounts of fine-grained computation, since automatic parallelisation of these types of programs tends to result in excessive overheads relating to management of parallelism.

Workflows fit within the first of these types of programs. Most external tasks invoked by a typical workflow involve a significant amount of computation, and already incur overheads associated with communicating with other machines across the network. In these situations, the small amount of additional book-keeping work involved with managing parallelism is insignificant when compared with the overall computation involved with the workflow. Parallel functional programming research, on the other hand, has historically only addressed programs which perform all their computation internally, without invoking external programs or services. For these types of programs, annotations have been found to be most practical, so existing parallel functional languages require programmers to explicitly annotate parallelism within their programs.

Since we are focused on workflows rather than internal computation, raw computation performance is not a critical factor in overall execution time. The execution efficiency of local code only needs to be fast enough to coordinate external services, and to perform small amounts of intermediate computation and data manipulation between calls. Automatic parallelisation is thus an appropriate choice, even if it does involve a certain degree of overhead for local code execution.

4.3.2 Sparking

A *spark* is a closure that has not yet begun evaluation, but whose result is known to be needed at some point in the future. Parallel functional language implementations typically employ a *spark pool* containing a set of sparks, which idle processors consult when looking for work. During execution, new closures are sparked when it can be determined that their result will be needed by the program. This can occur due to annotations added by the programmer, or by cases in which the compiler has been able to identify such expressions automatically. In either case, instructions within the compiled code control the sparking, such that the runtime system sparks specified graph nodes when these instructions are encountered.

Other parallel functional language implementations use this scheme for internal parallelism, and in such cases it is only useful when there are multiple processors involved with graph reduction. However, in the case of external parallelism, sparking is still useful even when the graph reduction is being performed on the client by only a single processor. The reason for this is that when a call is made to a remote service, the frame that made the call will block, and the processor will become idle. If there are other sparks available in the spark pool (representing either local computation or other service calls), these can begin execution while the other remote call is in progress.

NReduce considers frames and closures to be the same thing, and thus the way in which the data associated with a frame is stored in memory is independent of whether or not it has begun executing. Each frame has information associated with it indicating whether or not it is sparked. Additionally, all sparks are maintained in a spark pool, which is stored as a doubly-linked list. In order to spark a frame, it is necessary to check the state of the frame, and if it is not yet sparked, update the state and add it to the spark pool. When a frame begins evaluation, it is removed from the spark pool. We have optimised these operations to minimise the overhead associated with adding and removing frames from the spark pool, since these operations occur frequently during execution. Sections 6.3 and 6.4 cover the spark pool implementation and performance considerations in more detail.

To illustrate how sparking occurs, consider the following function, which computes the nth Fibonacci number for all integer values $n \geq 0$:

```
nfib n = (if (<= n 1)
             n
             (+ (nfib (- n 2)) (nfib (- n 1))))
```

This function is deliberately implemented in an inefficient manner in order to generate a large amount of parallelism. The + operation requires both of its arguments to be

```
Stack Addr Opcode     Arg0 Arg1
size

1      6   GLOBSTART   71   1    // Start of function #71
1      7   SPARK       0    0    // Spark parameter n
1      8   EVAL        0    0    // Wait for evaluation of n to complete
1      9   PUSHNUMBER  1         // Push 1 on to stack
2     10   PUSH        0    0    // Push n on to stack
3     11   JCMP        3    <=   // Jump to 14 if n <= 1
1     12   PUSH        0    0    // Push n on to stack
2     13   RETURN      0    0    // Return n
1     14   PUSHNUMBER  1         // Push 1 on to stack
2     15   PUSH        0    0    // Push n on to stack
3     16   BIF         -         // Subtract 1 from n
2     17   MKFRAME     nfib 1    // Create frame (nfib (- n 1))
2     18   SPARK       1    0    // Spark frame
2     19   PUSHNUMBER  2         // Push 2 on to stack
3     20   PUSH        0    0    // Push n on to stack
4     21   BIF         -         // Subtract 2 from n
3     22   CALL        nfib 1    // Call (nfib (- n 2))
3     23   EVAL        2    0    // Evaluate result
3     24   EVAL        1    0    // Wait for (nfib (- n 1)) to complete
3     25   BIF         +         // Add results of recursive calls
2     26   RETURN      0    0    // Return result
```

Figure 4.4: Bytecode generated for the `nfib` function

evaluated, and thus the compiler will recognise that sparking can be used to evaluate both in parallel. The bytecode generated for this function, shown in Figure 4.4, contains a SPARK instruction which sparks the `(nfib (- n 1))` call. The code then makes a direct call to `(nfib (- n 2))`. It would be possible to instead spark both expressions and then have this function block on their completion (using EVAL), but in this particular case the direct call has been added by the compiler as an optimisation.

4.4 Frame management

Each frame has a *state* associated with it, indicating where it is in the life cycle depicted in Figure 4.5. The state of a frame is either *new*, *sparked*, *runnable*, or *blocked*. A new frame represents a closure/function call node that has not yet begun evaluation. If lazy evaluation is being used, the frame may or may not begin evaluation at some point in the future; strict evaluation always causes new frames to be evaluated eventually. A sparked frame is one that has been identified as being needed in the future, and can usefully begin

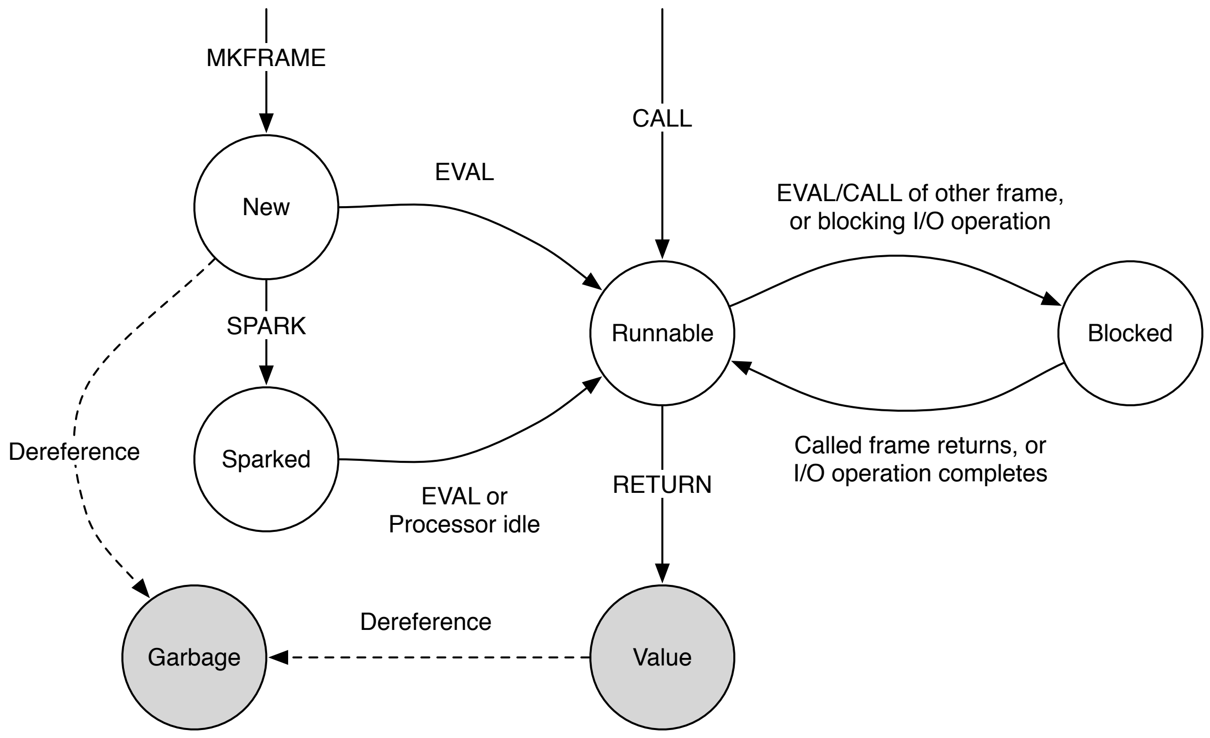


Figure 4.5: Life cycle of a frame

evaluation if the processor becomes idle. A frame in the runnable state has work to do immediately, and may be selected for execution by the processor. A blocked frame is one that is waiting for either another frame to return, or an I/O operation to complete.

The shaded states in Figure 4.5 represent states in which the frame no longer exists as a frame, but has been transformed into either a value (due to the frame returning), or memory that is ready to be reclaimed by the garbage collector (due to the frame or value becoming dereferenced). A frame can only become dereferenced when it is in the new state; if it has been sparked or has begun evaluation, this implies that its result must be needed by some other part of the program, which will maintain a reference to the frame until it returns.

There are two ways in which a frame can be created. The MKFRAME opcode creates a new frame, copying a set of arguments into its data area, but does not immediately begin evaluating it. Instead, the frame is placed in the new state, corresponding to a closure whose evaluation may potentially be triggered at some later point in time if its result turns out to be needed. The CALL opcode creates a frame in the same manner as MKFRAME, except that the frame immediately becomes runnable, and the calling frame blocks until the new frame returns. CALL is equivalent to MKFRAME followed by EVAL, but is more efficient, due to the fact that some intermediate operations, such as checking the type and status of the frame object, can be skipped.

During execution, a frame may switch between the runnable and blocked states multiple times. A frame blocks whenever it makes a call to a user-defined function, in which case it will only resume once that function returns. This is equivalent to the notion of stack frames that are below the top of the stack in an imperative language. A frame can also block if it makes a call to an I/O operation such as `read`, in which case it will resume once the operation is complete. The way in which context switches are made between frames is described in the next section.

A frame switches between the runnable and blocked states on every function call or return. To avoid the cost of updating the frame's state field whenever these events occur, a single value is stored in the field to indicate that the frame may be in either one of these two states. There are no situations that arise at runtime where it is necessary to determine whether a given frame is runnable or blocked. The linked lists maintained by the context switching mechanism described in the next section are sufficient to record information that can be used to determine the set of runnable frames, or the set of calling frames blocked on a particular callee, which is all that is needed during execution.

4.4.1 Blocking and context switches

A context switch is when the processor stops executing one frame, and starts executing another. It happens whenever a function call or return occurs, or a blocking I/O operation begins or completes. Context switches in NReduce are conceptually similar to those which occur between processes in a multi-tasking operating system, but involve much less overhead as they are done entirely in user space and involve fewer state changes.

During execution, the set of runnable frames is maintained using a singly-linked list, which is updated whenever a frame enters or leaves the runnable state. Whenever there are one or more frames in the list, the one at the front is always the frame that is currently executing. Because the link to the next frame in the runnable list is stored as a field of a frame object, a reference to the current frame doubles as a reference to the list. An empty runnable list implies that the processor is idle.

Each frame maintains a singly-linked *waiting list*, containing references to frames that are blocked on its completion. When a function call is made, the caller is added to the waiting list of the callee. If this call is made using the CALL instruction, which creates a frame and begins execution of it immediately, then there will only be one caller in this list. If the frame instead started execution as the result of an EVAL instruction, then it is possible that other frames may be subsequently added to the waiting list if they later call EVAL on their own reference to the frame.

Figure 4.6 shows the changes that are made to the runnable list and callee's waiting list when a function call and return occurs. The first thing that happens is that the caller is removed from the runnable list. Then, the callee is added to the start of the runnable list, becoming the new current frame. Finally, the caller is added to the waiting list of the callee. Upon return, the reverse happens — the callee is removed from the runnable list, and then each caller is added to the start of the runnable list and removed from the waiting list.

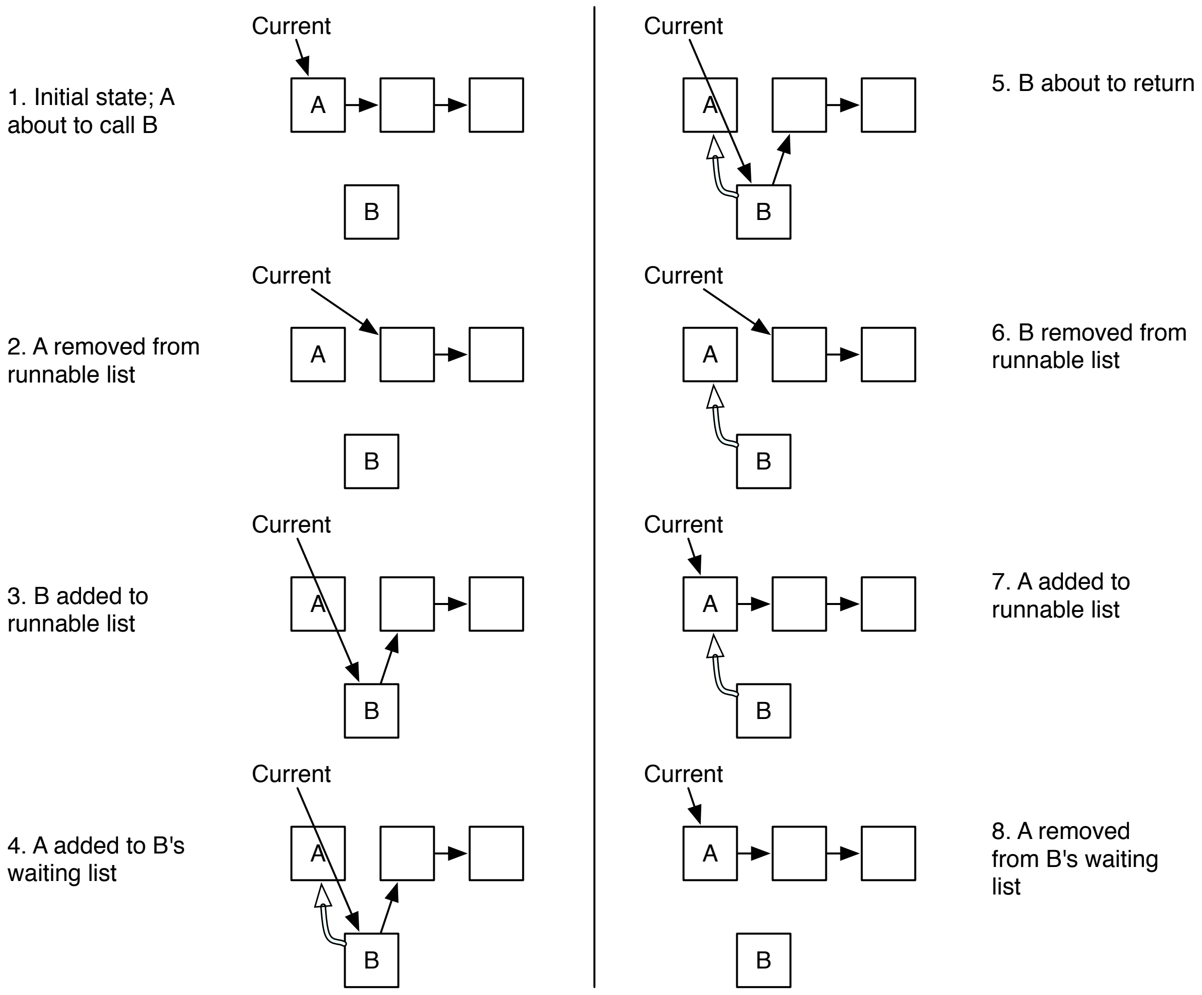

Current
1. Initial state; A about to call B
A
B
2. A removed from runnable list
3. B added to runnable list
4. A added to B's waiting list
5. B about to return
6. B removed from runnable list
7. A added to runnable list
8. A removed from B's waiting list

Figure 4.6: Context switching during function call/return

A normal function call/return sequence thus incurs the cost of six linked list operations, and is therefore substantially more expensive than in stack-based language implementations, which typically just adjust the stack pointer and push/pop the base pointer. Although this contributes to NReduce's slower performance than compiled imperative languages like C and Java, the testing described in Section 7.6.1.4 indicates that the function call performance is still well ahead of other mainstream scripting languages like Perl and Python. This level of performance is adequate for our purposes, since ELC is designed for only lightweight computation, much like other scripting languages.

4.5 Networking

Functional programming, in its purest form, concerns only data transformation, not state manipulation. However, existing functional languages expose network connections in a stateful manner, forcing an imperative style of programming to be used when performing network access, as discussed in Section 2.4.6. Data is transferred using individual read and write operations, each of which manipulate the state of a connection and must therefore be carried out in a sequential manner. While this is appropriate for applications that do actually involve state, it unnecessarily sacrifices the potential for exploiting parallelism in situations where the network interaction only involves calls to side effect free services.

In Section 3.5.8, we proposed an alternative approach to supporting network access, in which each connection is modeled as a function which transforms from input to output. In this approach, the programmer does not deal with connection objects directly, but instead supplies the input data to be sent to the service, and receives the output as the result of the function call. This is a purely data-oriented view of service access, which does not require the programmer to be concerned with low-level stateful operations. It also opens up the possibility of invoking multiple service calls in parallel, due to each being treated as a pure, side effect free function.

The implementation of this abstraction involves several different parts of the virtual machine. Processing of different data streams is performed by concurrently-evaluated expressions, with frames from each blocking and resuming independently. Built-in functions are provided for performing reads and writes on connection objects, which are used internally by wrapper functions exposed to the programmer that hide the state manipulation required to send and receive data across the connection. An I/O thread runs concurrently with the execution thread, to manage connections using system calls provided by the operating system. These aspects all interact to provide the stream-based mechanism that the programmer sees.

Data streams are simply lists of bytes, and are manipulated using standard functional list processing operations such as `cons`, `head`, and `tail`. As far as the programmer is concerned, all lists appear as chains of cons cells. However, an array-based representation is often used internally, which stores list elements as arrays of bytes in memory. This representation uses less memory, and is more efficient for data transfer, since read or write operations on connections can deal with chunks of data instead of individual bytes. This mechanism is described further in Section 4.6.

4.5.1 Abstracting state manipulation

The wrapper functions used to implement the data-oriented connection abstraction are defined in ELC, and made accessible to all programs. These make use of built-in `read` and `write` operations provided for use with connection objects, which are supported as a special type of graph node by the virtual machine. These functions and objects are purely internal implementation details, and are never accessed directly by programmers.

From the perspective of user code, network connections are always accessed through the `connect` function. This takes parameters specifying the host and port to connect to, as well as a data stream to be sent across the network. `connect` uses two helper functions to achieve this: `writeloop`, which traverses the input stream and writes its contents to the connection, and `readloop`, which reads data from the connection and produces the output stream. The code for these functions is shown in Figure 4.7.

```
connect host port out =
(letrec conn = (opencon (lookup host)
                        port
                        (writeloop conn out))
 in (readloop conn))

readloop conn =
(read conn (readloop conn))

writeloop conn stream =
(if stream
  (letrec partlen = (nchars stream)
   in (if partlen
        (seq (writearray conn partlen stream)
             (writeloop conn (arrayskip partlen stream)))
        (seq (write conn (head stream))
             (writeloop conn (tail stream)))))
  (writeend conn))
```

Figure 4.7: Wrapper functions for accessing connections

The built-in function `opencon` is what actually opens the connection. It first requests the I/O thread to make an asynchronous `connect` system call[1], and then suspends the current frame. When the I/O thread detects completion of the call, it notifies the execution thread, which resumes the frame. The frame then invokes `opencon` a second time, which inspects the result of the system call to determine whether or not the connection was established successfully. If so, the function will create and return a new connection object, which can subsequently be used to read or write data.

Upon success, `opencon` causes its third parameter, a closure representing a call to the `writeloop` function, to begin evaluation. `writeloop` traverses the input stream, and as data becomes available, writes it to the connection. On each iteration, the function checks the current portion of the input stream to see if it is stored using the optimised array representation discussed in Section 4.6. If it is, the data is written to the connection as a chunk containing however many bytes of data are in the array. Otherwise, if the list is stored using cons cells, the data is written one byte at a time. We use the built-in `seq` function, which forces its first parameter to be evaluated prior to its second, to ensure that the current portion of the input stream is written before the next. When the end of the stream is reached, the built-in function `writeend` is called, which performs a write shutdown on the connection object.

The internal logic for reading from the connection resides in the built-in `read` function,

[1]The `connect`, `read`, and `write` functions in Figure 4.7 are distinct from the system calls of the same name.

which is invoked by `readloop`. The second parameter to `read` is a closure representing another call to `readloop` which will read the next portion of the stream. A single call to the built-in `read` function causes a request to be sent to the I/O thread to obtain the next chunk of data from the connection. As soon as this data becomes available, the execution thread is notified and supplied with the data. This data is then returned as an array, the tail of which refers to the closure representing the next call to `readloop`. The `read` function is non-strict in its second argument, as the recursive call can only be made after the current portion of the output stream has been read.

4.5.2 The I/O thread

The standard way to access network connections under most operating systems is the BSD socket API [283], which provides facilities for both blocking and non-blocking operation. Simple programs that only deal with one connection at a time typically use these APIs in blocking mode. Programs dealing with multiple connections have two choices: they can use a separate thread for each connection and make blocking calls for each, or they can use a single thread which uses non-blocking calls, and an event loop to determine when each connection is ready to be processed. The latter approach scales better, because it permits a large number of connections to be handled without incurring the cost of a thread for each.

One problem with the event-driven approach is that the thread cannot easily do other work while it is waiting for an event. If it does have other computation to perform, then it can spend its time doing this, while periodically polling to check if an event has occurred. However, this has the disadvantage of complicating the computation logic by requiring it to be intermixed with I/O processing code, and also introduces a delay between the time at which an event occurs and the time at which the program responds to it.

We deal with this problem by using two threads: one for execution, and another for I/O. The execution thread spends all its time performing computation, as long as there are runnable frames available. The I/O thread operates in an event loop, spending most of its time blocked in the `select` system call. When a connection becomes eligible for reading or writing, or an asynchronous `connect` system call completes, the I/O thread is woken up and takes the appropriate action, such as reading from the connection. It then notifies the execution thread by sending it a signal, and making the result of the operation, such as the newly received data, available as part of a message.

The execution thread's signal handler does not process the message immediately, but instead arranges for execution to be suspended once the current bytecode instruction has completed. This is to ensure that the handler routine sees the graph in a consistent state, in case it was part way through being modified when the signal was received. When execution is eventually suspended, the notification message is handled by taking the appropriate action, such as resuming a blocked frame and placing the received data in a heap object to be returned from a function. The same approach to interruption and message handling is also used for communicating with execution threads on other machines when the virtual machine is running in distributed mode, as described in Section 4.7.2.

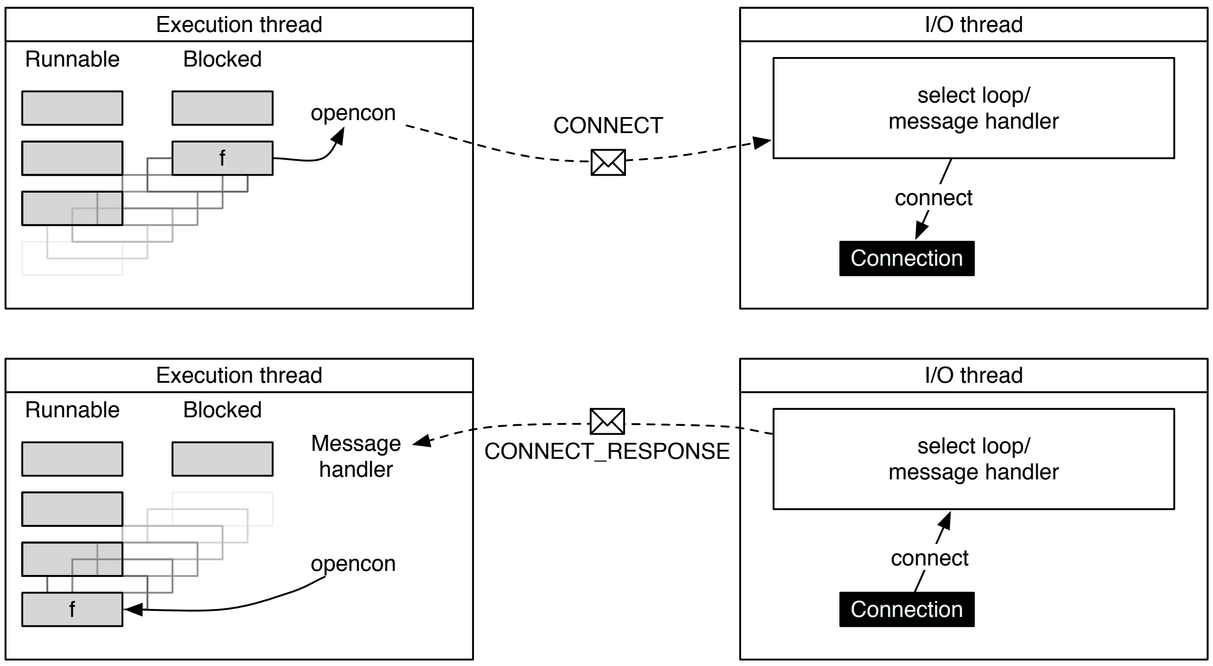


Figure 4.8: Request and response messages for an asynchronous I/O operation

4.5.3 Asynchronous call example

Figure 4.8 illustrates the above interaction in more detail by showing the steps involved with an asynchronous invocation of the `connect` system call. The first call to the built-in `opencon` function causes the current frame to be suspended, and a CONNECT message to be sent to the I/O thread. When this message is received, it causes the I/O thread to awake from its `select` loop, and initiate an asynchronous invocation of the `connect` system call. While the connection establishment is in progress, the frame that called `opencon` resides in the blocked state. Other frames may be on the runnable list however, and these can continue executing.

While a connection is being established, the socket that is being connected is included in the set passed to the `select` system call. When the operation completes, `select` returns, indicating that the socket is ready to be processed. The I/O thread inspects the result of the call, and then sends a CONNECT_RESPONSE message back to the execution thread indicating the result. An identifier supplied as part of the original CONNECT message is sent back in the response, so that the execution thread is able to correlate this response with the frame that made the request. This frame is then placed back on the runnable list, and continues execution from the point at which it was suspended. The first instruction that the frame executes is the same as the last one it executed previously, which is to call `opencon`, which this time creates the connection object. Other I/O operations, such as `read` and `write`, follow this same pattern.

A program may have multiple I/O operations outstanding at the same time. For example, one frame may be part-way through opening a connection, while two others are attempting to read from other connections, and yet another frame is trying to write to a connection. Because the frames may block and resume independently, each of the connections may be handled separately. This means that multiple service calls can be in progress at the same time, without the programmer having to concern themselves with the details of how this concurrency is achieved.

4.6 List representation

Most functional languages represent lists using cons cells, each of which has a *head* and a *tail* field, and acts as a node in a linked list. Functions such as `cons`, `head` (or `car`), and `tail` (or `cdr`) can be used to construct a wide range of list processing algorithms, often using recursive functions to perform traversal. This is a simple and convenient mechanism that is well known in the functional programming community.

Some functional languages, such as Haskell, represent strings as lists of characters. This has the advantage of not requiring a separate library of functions for string manipulation, because it is possible to instead use functions defined for other types of lists. We chose to use this approach, in order to keep our programming model simple. We also use this same representation for text and binary data transferred over network connections.

One of the drawbacks of cons lists, however, is efficiency. Large linked lists are significantly more expensive than arrays, because each element requires a separate cons cell, and elements are not stored in contiguous locations in memory. This is particularly relevant when performing I/O operations, which in other languages typically involves reading or writing arrays of bytes to a file descriptor. I/O using cons lists requires data to be processed one byte at a time, which is slow.

One way to mitigate this problem would be to expose arrays explicitly in the programming model, and provide functions for their creation and manipulation. While some functional languages take this approach, we instead wanted to keep the programming model as simple as possible, extending the basic lambda calculus model only where absolutely necessary to achieve efficiency. For this reason, we developed a method of supporting arrays in a manner which achieves the benefits of efficiency, but does not require them to be explicitly dealt with by the programmer.

The basic approach we use is to support arrays as a special data type that is hidden from the programmer. We implement all of the basic list functions in a polymorphic manner, such that they work with either an array or cons list, and produce the same results regardless of which is used. Cons lists are converted to arrays whenever possible behind the scenes, so the programmer never has to manually perform conversion, or even be aware of the difference. This means that as far as the programming model is concerned, every list appears as a sequence of cons cells, but in reality the virtual machine can sometimes store them as arrays.

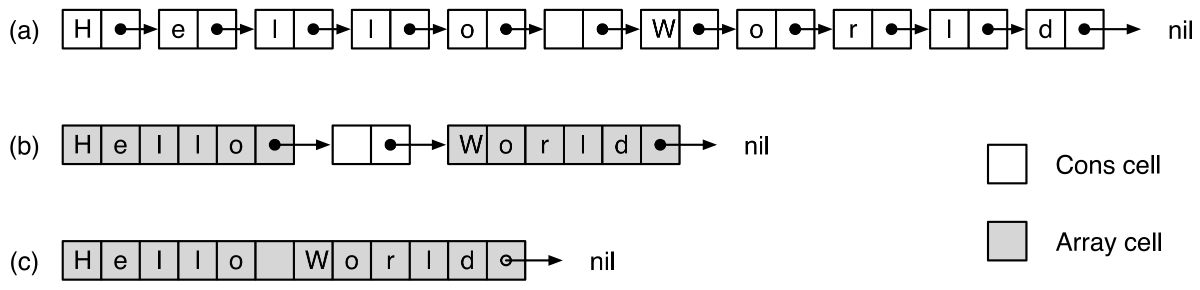


Figure 4.9: Three possible representations of the string “Hello World”

4.6.1 Array cells

In our implementation, array cells are essentially like cons cells which can store multiple list items instead of just a single item. Like cons cells, they have a tail field pointing to the next portion of a list. This means that a list can be constructed out of a combination of array cells and cons cells, linked together via their tail pointers.

One of the main benefits of this approach over forcing all of a list to be stored in array is that it can be used with lazy evaluation. It is possible for lazy functional programs to create lists in such a manner that at a given point in time, only part of the list may be materialised in memory, with the tail of the last cons cell pointing to a closure which can be used to compute the rest of the list. By maintaining this model of linked objects within the context of arrays, lazy evaluation can still be used in this manner. It also permits efficient prepending of items onto a list, since it is not necessary to modify the portion of the list that is being prepended to.

Figure 4.9 shows three possible representations of the string “Hello World”. Representation (a) uses a separate cons cell for each character. Representation (b) uses a combination of both arrays and cons cells; the substring “Hello” is stored in one array, followed by a cons cell for the space character, and another array for “World”. Representation (c) uses a single array for all elements in the list. In all cases, the last tail pointer is set to *nil*, indicating the end of the list.

The programmer is never made aware of which portions of a list are stored as arrays or cons cells. They merely access the list using the built-in functions `cons`, `head`, and `tail`, as well as higher-level library functions such as `item` and `len`. As far as the list appears to the programmer, each item is stored in a separate cons cell, and references can be made to any of these within the list. In the case of arrays, an *array reference* is used instead of a pointer to a cons cell. This requires the use of a special pointer representation, discussed in the next section.

4.6.2 Array references

When using lists built out of cons cells, a program is able to reference a particular portion of a list via the corresponding cons cell. A call to `head` on a cons cell returns the element

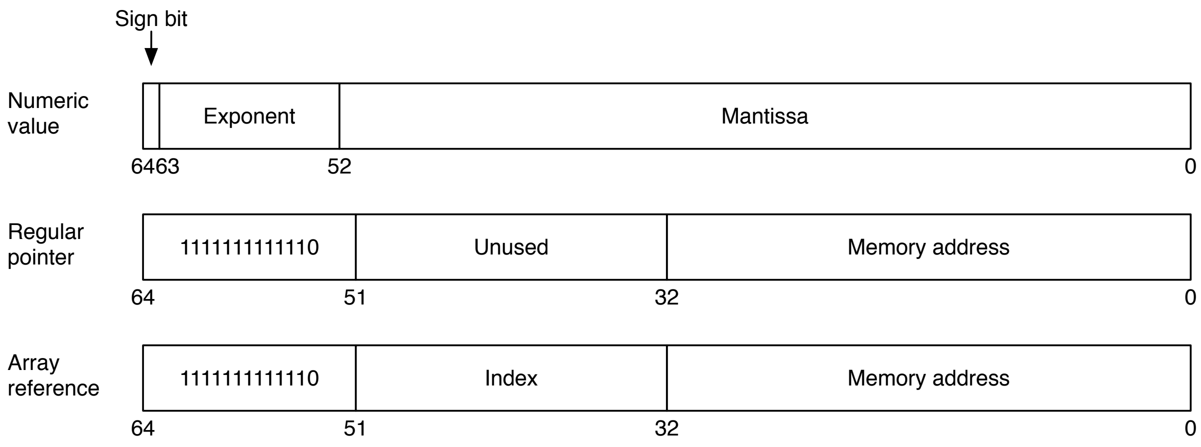


Figure 4.10: Encodings for array references, pointers, and numbers

at that position in the list, and a call to `tail` returns the next cons cell in the list. Since our array representation is intended to be transparent to the programmer, the `head` and `tail` functions need to provide the same semantics. An array can contain multiple elements though, so by itself, a normal pointer to an array cell does not contain enough information to indicate which position in the array it refers to.

In order to support references to positions *within* arrays, we use a pointer representation that stores not only the memory address of the array cell, but also the index within that array. Our use of 64-bit pointers within the context of a 32-bit memory model leaves additional space in the pointer to store this index value, since only 32 bits are needed for the actual object address. This enables all array references to include the position within the array that they refer to, and allows the the `tail` function to return a pointer to the same array but with a different index.

Figure 4.10 compares the encoding of array references to regular pointers and numeric values, which use the representation described in Section 4.2.4. Like regular pointers, array references are encoded such that they appear as NaNs when interpreted as floating point numbers, enabling them to easily be distinguished from regular numbers. This encoding uses the top 13 bits to indicate that the value is a NaN, leaving 32 bits to store the address of the array and 19 to store the index. This scheme makes it possible to reference up to 2^{19} = 524,288 different elements in an array. If more elements than this are present in a list, then they are stored across multiple array cells.

Whenever an array reference is used, both the array cell and index are extracted from the pointer. A call to `tail`, for example, will check if the index is before the end of the array, and if so, will return another reference to the same array but with a different index. A call to `head` will use the index to determine which element to fetch from the array. An array reference can thus be used in the same manner as a cons cell, and in fact the programmer is never even aware of the difference.

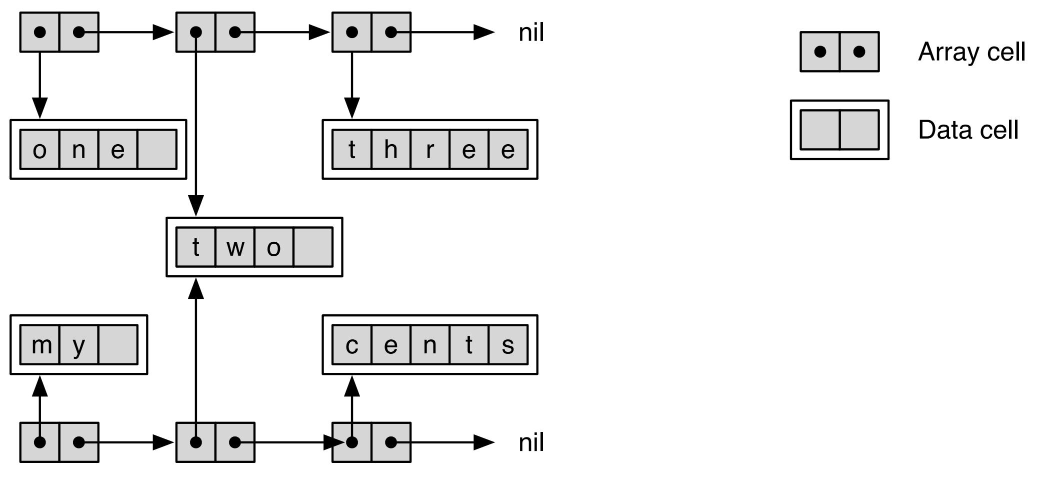


Figure 4.11: Multiple lists sharing common data

4.6.3 Sharing

With cons lists, it is possible to create multiple lists that share the same suffix, by arranging for the tail field of the last cell in the unique portion of each list to point to the first cons cell in the shared portion. This can also be done for arrays, which are linked together in the same manner. When using arrays however, it is also possible to create multiple lists which contain shared data part-way through, and end with different suffixes.

Although Figure 4.9 showed arrays as if they directly contained their data, in reality the contents of an array are stored in a separate *data cell*, which is distinct from the array cell itself. It is possible to have multiple array cells referring to the same data cell, in which case the lists that contain those array cells share the contents of the data cell. Each array cell can have a different tail field. Sharing occurs when the `++` (concatenate) function is passed an array cell as its first parameter. It creates a new array cell referring to the same data cell as the first parameter, with the tail pointer initialised to the second parameter.

For example, suppose there was a variable called `two`, bound to the string "two "; the following expressions would produce the array and data cells shown in Figure 4.11.

```
(++ "one " (++ two "three"))

(++ "my " (++ two "cents"))
```

For programs which contain many lists that share large amounts of data but have different prefixes and suffixes, this reduces the amount of memory required. Even more importantly, it reduces the amount of data that must be transferred across the network when the virtual machine is running in a distributed manner. The reason for this is that when multiple lists need to be copied from one machine to another, the shared portion only needs to be transferred once.

4.6.4 Converting between representations

Because arrays are not exposed as part of the programming model, there is no way for a programmer to create one explicitly[2]. Instead, lists must be constructed in the usual manner using `cons`. When the runtime system notices that a list of cons cells is present in memory, it automatically converts these into an array. This check is performed every time a cons cell is accessed, so that a newly constructed cons list will be transformed into an array as soon as it is accessed. Checks are also performed on the tail fields of arrays, so that if it turns out that the tail refers to another cons cell, it will be merged into the array.

Another conversion takes place when it is detected that all of the elements in the list are evaluated, and are whole numbers in the range 0-255. This is true of ASCII strings, as well as binary data. When this condition is met, the array is converted to a more efficient representation in which only one byte per element is used. Arrays produced by read operations on network connections also use this format. This compact representation makes it possible for programs to deal with large amounts of text or binary data while still being subject to the illusion that this data is a cons list which can be manipulated using normal list operations.

4.6.5 High-level list operations

Although arrays use less memory than cons lists, there is no efficiency gain if they are only accessed using linked list operations. For this reason, NReduce provides a library of high-level list operations which work with both cons cells and arrays, but operate more efficiently for the latter. These operations include length calculation, indexed element access, concatenation, and subsequence extraction.

All of these are functions which could be implemented in terms of the basic `cons`, `head`, and `tail` primitives. However, doing so would not take advantage of the efficiency gains possible with arrays, since these functions only deal with individual elements at a time. The high-level library functions make use of additional built-in functions which allow various operations to be performed more efficiently on arrays. These functions are purely an optimisation, designed to let programmers take advantage of arrays without having to use a separate set of functions to those used for cons cells.

4.7 Distributed execution

Two important styles of workflow execution are *orchestration* and *choreography*, introduced in Section 2.3.4. Most workflow engines use orchestration to invoke tasks, whereby the workflow engine runs on a single machine and communicates with services in a client-server fashion, as described in Section 3.4.1. Each time a service is invoked, the input

[2]With the exception of string literals, which are arrays of characters.

parameters are sent to the service, and the result is sent back to the client. If the result of a service call then needs to be passed as a parameter to another service, this data gets transferred from the client to this other service.

When large amounts of data are passed between services, choreography can be used to reduce data transfers by arranging for a result produced by a service to be transferred directly to the site at which it is consumed. However, most services only understand request/response messages, and lack the ability to send results to any machine other than the one which sent the request. Choreographing such services requires additional software to be placed on the machines which host the services, to take care of transferring result values directly to their destination site, instead of the one which supplied the input data.

Section 3.4.2 described the idea of a *distributed* workflow engine/virtual machine, which consists of instances running on each machine that participates in the workflow. Each instance takes care of executing part of the workflow, and acts as a client to invoke service calls on the local machine via request/response messages. Any data transfers that need to take place between services on different machines go through the instances of the workflow engine, which communicate with each other to coordinate the overall execution of the workflow. Figure 3.2 on page 92 showed an example of this in a configuration of four nodes.

In this section, we illustrate how this style of service choreography can be achieved using distributed memory parallel graph reduction.

4.7.1 Architecture

Figure 4.12 shows how the virtual machine can be deployed in a distributed manner for the purposes of service choreography. The nodes maintain connections to each other, and use a message passing protocol to perform the interactions involved with parallel reduction of the graph. An object-based distributed shared memory mechanism allows the graph to be automatically partitioned across the nodes, each of which performs reduction on its own portion of the graph. A separate spark pool is maintained for each node, containing sparked frames which are candidates for distribution to idle machines.

Each machine also runs instances of the services that are to be used by the workflow. These can be provided by any regular service hosting environment, for example web service containers such as Axis [251] and WebSphere [320]. The services themselves know nothing about the workflow engine, and accept connections from clients using regular TCP-based protocols such as HTTP. Invocation of service operations occurs using the standard mechanisms, but occurs by having the local instance of the workflow engine make the request, instead of one running on a separate machine.

4.7.2 Message passing

Distributed execution involves each node running an execution thread, which performs bytecode execution using the frame and graph structures described in earlier parts of this

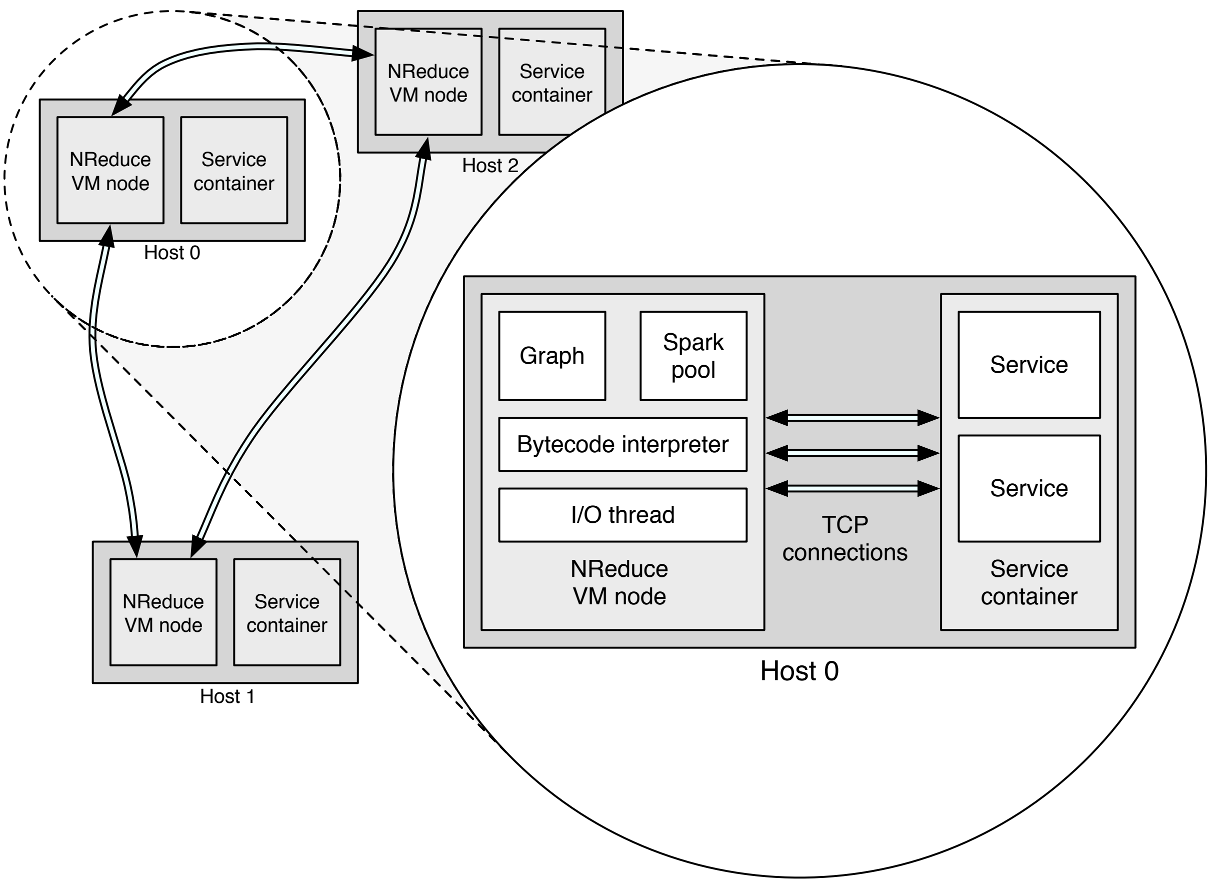


Figure 4.12: Distributed virtual machine with co-hosted services. The zoomed-in part of the diagram shows the components of an individual virtual machine node.

chapter. Additionally, other threads are used for support facilities like distributed garbage collection, process startup, and network I/O. All of these threads interact with each other using a message passing facility provided by the virtual machine, which supports both local and remote communication. This facility is based on an asynchronous, one-way messaging model, inspired by the message passing semantics of Erlang [21].

The main reason for using asynchronous communication is that most messages sent during execution are requests for graph cells stored on other nodes. When such a request is made, it is only necessary for the frame that needs that value, rather than the whole execution thread, to block. If other frames are in the runnable list, they can continue executing while the blocked frame is waiting for a response. This is similar to the approach used for concurrent processing of network connections discussed in Section 4.5.2.

The two basic operations provided for message passing are `send` and `receive`. Each thread is addressed using a combination of the node's IP address and a local identifier. A `send` operation specifies the thread's address, a message tag, and a payload. The operation returns immediately, and the message transfer is handled in the background by the I/O thread, which buffers the outgoing message data and writes it to the connection as soon as possible.

The `receive` operation can operate in either blocking or non-blocking mode. A blocking `receive` causes the calling thread to be suspended until a message arrives, or a specified timeout elapses. This is used for threads which operate in an event loop, and have no work to do other than responding to messages. A non-blocking `receive` simply returns immediately if no message is available yet.

Threads which have other work to do in addition to responding to messages, such as an execution thread with one or more runnable frames, can chose to be notified of message arrival by arranging for the I/O thread to send a signal to the thread when a message arrives. This allows the thread to spend its time performing computation, without having to poll for message arrival at regular intervals. In the case of the execution thread, the signal handler arranges for the message to be handled as soon as the current bytecode instruction finishes executing. This ensures that the message handling routine does not see the graph in an inconsistent state, due to it having been part way through modification when the signal was received.

Although there is no way to check the result of a `send` operation, a thread can chose to be notified asynchronously when a communication error is detected. It does this by registering a *link* for each other thread it is interested in. The communications layer monitors all connections to other nodes, and when it detects an error on one of them, it sends endpoint failure messages to threads that have registered their interest. This enables errors to be handled as a particular message type within an event loop.

4.7.3 Distributed heap management

During distributed execution, each node stores the portion of the graph on which it is performing reduction. Each cell in the graph is stored on at least one machine, with others maintaining replicas where appropriate. Because cells on different nodes may reference each other, it is necessary to support both remote and local references, so that the structure of the graph can be maintained. Parts of the graph that reside on one machine may be needed by frames running on another, so it is also necessary to provide mechanisms to allow values to be requested and transferred. To manage the distributed storage of the graph, we use the model originally proposed by Trinder et. al. for the GUM runtime system [300], which the remainder of this section describes.

Within a particular node, portions of the graph that are stored locally use the same memory representation and referencing mechanism as for local execution. References to a local cell use the pointer encoding described in Section 4.2.4, which stores the memory address of the target cell. However, references to cells on other machines must be stored differently, as more information is required to identify a cell on a remote machine. Accessing the target cell associated with a remote reference requires a request/response message interaction, so that a local copy of the cell can be obtained.

Remote references go through an indirection layer, which associates a *global address* with each globally-accessible cell. A global address consists of the node number and a local identifier for the cell, the latter of which is unique within the context of a particular

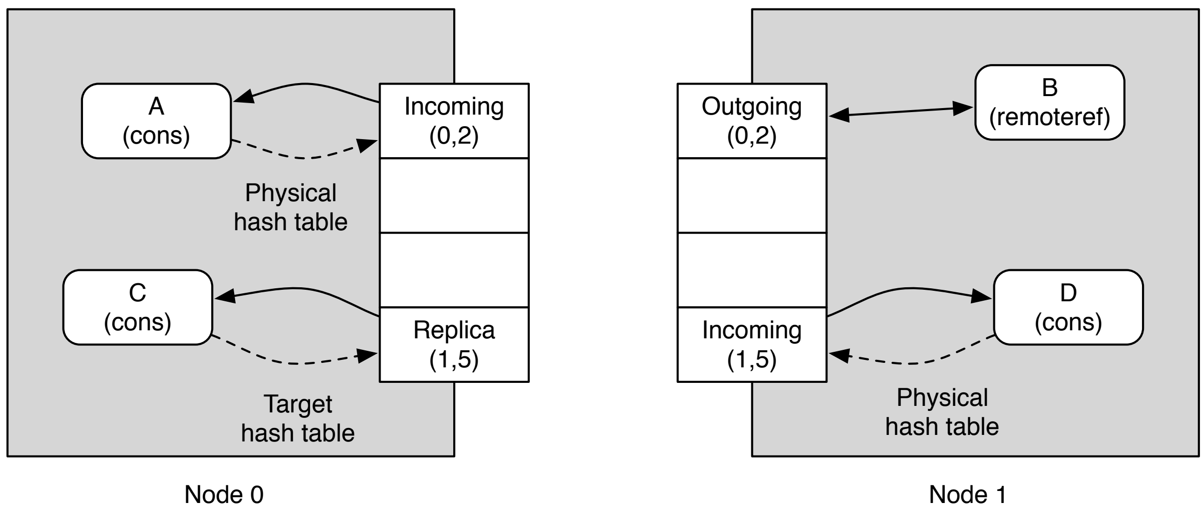


Figure 4.13: Global address table

node. The reason for using a separate identifier instead of the actual memory address of the cell is because it is possible that the memory address may change as a result of copying performed by the garbage collector. Global addresses are passed around within messages exchanged between nodes whenever a message needs to refer to a cell on one of the nodes. Each node maintains a *global address table* (GAT) containing information about globally-referenced local or remote cells.

There are three types of entries that may exist in the global address table. An *incoming reference* refers to a local cell that is referenced from at least one other node. An *outgoing reference* refers to a cell residing on another node which is referenced from the local heap. A *replica* refers to a cell which resides on another machine, but for which there is a local copy that has been obtained as a result of a fetch request. For each of these types, the GAT entry stores the global address of the cell, as well as a pointer to a (possibly different) cell in the local heap. In the case of an incoming reference, the local cell is the same as the original. For an outgoing reference, the local cell is a *remote reference* cell containing a pointer to the GAT entry which can be used for lookups when the value of the cell is required by a frame. For replicas, the local cell is a copy of the remote cell.

Three hash tables are maintained to facilitate lookups into the global address table. The *address hash table* maps between global addresses and GAT entries, and is used when decoding messages received from other nodes. The *physical hash table* maps between local cell pointers and GAT entries, and is used for encoding a reference to a local cell when sending a message. The *target hash table* is used to maintain mappings between replica cell pointers and GAT entries, so that when a reference to a replica is sent, the message contains the address of the original cell instead of the local replica, enabling it to be identified properly by the recipient.

Figure 4.13 shows a graph distributed across two nodes, illustrating the different types of entries that can be present in a global address table. Cell A is a cons cell, which has been given the global address (0,2). An incoming reference entry for this cell is present in the global address table of node 0, as well as an outgoing reference entry in the table for node 1. In the latter case, the local cell associated with the entry is B, which is a *remote reference* cell containing a pointer to the entry. A does not have a direct pointer to its GAT entry; its mapping is instead maintained by the physical hash table. The reason for this is that cell types other than remote references do not contain fields for GAT pointers; these are only needed for a small portion of cells, and would be wasteful if given space in every cell type.

Cell D on node 1 is also a cons cell, and has an incoming reference entry similar to that of A. However, node 0 has a *replica* of this cell, instead of a remote reference. The replica is a copy of the cell which still maintains its association with the original, via a replica entry in the GAT. As with an incoming reference, there is no way to store a pointer directly from the replica to the GAT entry, due to there not being a field in the cons cell for this purpose. The mapping in this case is stored using the target hash table.

The reason for maintaining separate target and physical hash tables is that a cell can potentially be associated with two different global addresses. One is the physical address on the cell's local node, and the other is the target address, used for replicas and remote references. When a fetch request is sent to retrieve the target value associated with a remote reference, the physical address of the reference is included, so that it can be sent back with the reply. When the reply arrives, the runtime system uses this physical address to determine which remote reference cell to update with the value.

4.7.4 Work distribution

Some workflow systems use *static scheduling*, in which information is obtained about the available machines prior to execution, and used to produce a plan dictating when and where certain tasks will be executed. The problem with this approach is that it does not take into account dynamic load changes on the machines hosting the services. Many services are used by multiple clients, and in general it is impossible predict what impact these other clients will have on the load. We instead believe that *dynamic scheduling*, in which decisions about where to execute tasks are made on demand, is a far more practical approach.

Another advantage of dynamic scheduling in the context of our system is that unlike many other workflow systems, we use a Turing complete programming model. In the general case, it is impossible to determine ahead of time how long different parts of the workflow will take to execute, or even how many service calls will be made, since doing so would be equivalent to solving the halting problem. Workflows in our system can also make decisions about which services to invoke based on the results of previous service calls, meaning that even the set of services that are actually used by a workflow cannot be determined statically. This completely rules out the use of any static scheduling approaches.

Fortunately, the problem of dynamic scheduling for functional programs has already been addressed by other language implementations similar to ours. We decided to adopt an existing approach that others have found to be effective. As with the distributed heap management logic, we adopt the model for work scheduling/distribution used by GUM [300], which chooses when to distribute frames to different machines in order to achieve load balancing. Unlike GUM however, our implementation extends this idea to scheduling of not only internal computation, but also external service requests. The latter case serves as a demonstration of how this technique can be applied to workflows.

The work distribution mechanism is conceptually quite simple. Any time a node of the virtual machine is idle, it sends a FISH message to a random neighbouring node, representing a request for work. If the receiving node has one or more frames in its spark pool, it selects several of them and migrates them to the idle node. Otherwise, the receiving node will forward the FISH message on to another node, which will again check for sparked frames. The message continues propagating between nodes until sparked frames are found, or a time-to-live value associated with the request reaches zero. We chose this approach because it decentralises scheduling decisions among the participating nodes, rather than relying on a central point of control which could potentially become a bottleneck. The number of frames sent in response to a work request was derived experimentally, as discussed in Section 6.4.7.

When a frame is migrated ("scheduled") to an idle machine, it is immediately placed in the runnable state. An idle node will typically receive multiple frames as a result of work request, to avoid situations in which single frames which only have a small amount of work to do cause lots of frequent requests. When the frame is migrated, the associated graph cell on the source node is converted into a remote reference, preventing the frame from undergoing a redundant second evaluation on the source. If that frame's value is subsequently needed on the source node, a fetch request will be sent to the node which was given the frame, which will send back the result when the frame returns.

One aspect of this which deserves special attention for the case of workflows is how it is determined whether a machine is idle. It is possible that the virtual machine node may have an empty runnable frames list, but the machine may still be busy processing one or more service requests. For this reason, a given host is only considered idle if the runnable frames list is empty *and* the number of service requests in progress is below a certain threshold. The virtual machine keeps track of how many connections are open to local services, and avoids sending out work requests when this is at the threshold. Section 6.4.2 discusses this mechanism in more detail.

4.7.5 Frame migration

In addition to the work distribution mechanism described in the previous section, frames may also migrate when accessing a service. When a frame attempts to connect to a service on another machine, it migrates there, so that all interaction with the service occurs locally. One reason for this is that so that any input data required by the service which initially resides on another node will be transferred to the service node on demand, avoiding a potential second transfer if the service were to be accessed from the node on which the frame originally resided. A second reason is that in the case where the result

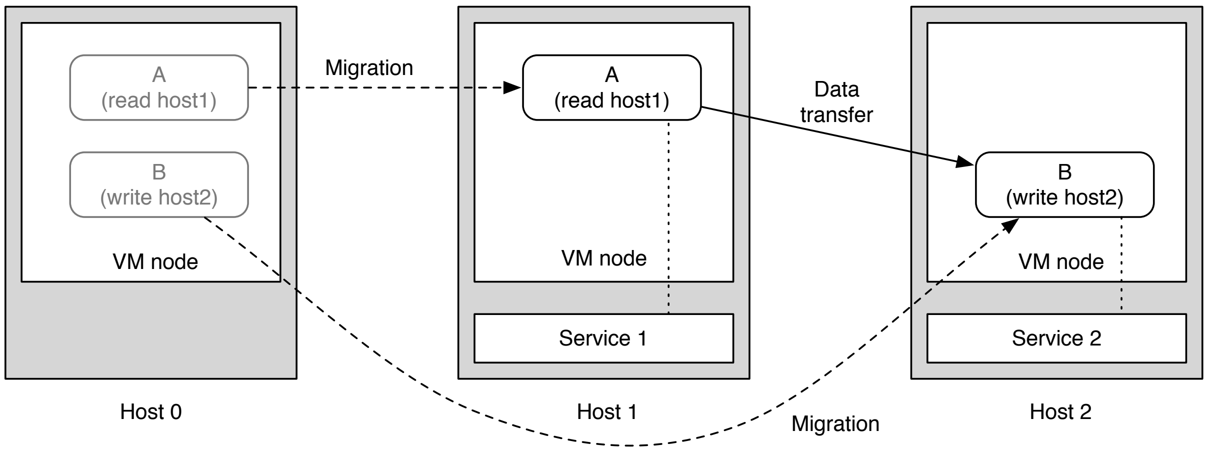


Figure 4.14: Frame migration triggered by service access

data produced by the service is required by another service on the same node, it does not need to be transferred across the network. The distributed heap management logic described in Section 4.7.3 takes care of the necessary data transfers.

Frame migration is triggered by the built-in I/O functions when they notice an attempt to access a connection on another machine. For `opencon`, this corresponds to a host parameter whose IP address is different to that of the current node. For other functions such as `read` and `write`, this occurs when accessing a replica of a connection object that resides on a different machine. In either case, the runtime system is instructed to migrate the frame to the host on which the connection resides, from where it will continue executing.

When migration occurs, only the frame itself is transferred, and not any cells that it references. This means that when execution of the frame resumes on the destination node, there may be variables within the frame's data area that it thinks are local cells, but instead are now remote references. If the frame tries to access one of these without first checking that the value is available locally, then this will cause an error to occur. We avoid this problem by restricting migration only to frames whose sole purpose is to evaluate their arguments and then invoke a built-in I/O operation; these can be safely restarted from the beginning, causing their parameters to be fetched as appropriate.

Figure 4.14 shows how this frame migration would enable direct transfer of service results. Three nodes are present in the virtual machine, two of which host services. The workflow is defined by the following code, which invokes service 2 with an input parameter obtained from the call to service 1:

```
f x = (connect "host2" 1234 (connect "host1" 1234 x))
```

Execution initially begins on host 0. The first `connect` call to execute is the one which accesses service 1, and causes its `opencon` frame to migrate to host 1, where a connection object will be created. This connection object is then passed to the `readloop` function call spawned by `connect`, which in turn passes it to the `read` frame. Since the connection is on host 1, `read` migrates there from host 0. The data obtained from the service is stored on host 1 once the `read` function returns.

The second service access, on host 2, follows the same pattern for establishing the connection. The `opencon` call which connects to host 2 migrates there and creates a connection object. The `writeloop` function makes a call to `write`, which migrates to host 2, where the connection resides. Since the data to be supplied to service 2 is obtained directly from the result of service 1, the `write` frame on host 2 sends a fetch request for the value of the `read` frame. This causes it to obtain the data directly from host 1.

4.8 Summary

This chapter has described the design and implementation of the NReduce virtual machine, an execution engine designed for running workflow-style applications expressed as functional programs. The virtual machine supports an input language, ELC, which is a variant of lambda calculus, and provides a small number of low-level primitives which can be used to build up abstractions for representing various types of data structures and network protocols. The main purpose of the virtual machine is to serve as a platform for running high-level workflow languages which are implemented by compilation into ELC, and example of which is our XQuery implementation described in the next chapter.

The way in which parallelism is supported within the virtual machine requires a fundamentally different approach to execution to what is used for imperative languages. Instead of relying on explicitly-defined threads, which the programmer must specify and control, the parallelism is handled implicitly by automatically sparking every possible expression whose value can be determined to be required. While this carries a certain overhead in terms of raw computation performance, it has the benefit of being very easy for the programmer to use. Because workflows delegate most of their compute-intensive work to external services, the performance cost associated with managing this parallelism comprises only a small portion of the application's total execution time, making the tradeoff acceptable.

A graph-based representation of the program state is used during execution, which has much in common with existing workflow engines, which are also based on a graph model. Rather than directly manipulating the graph however, the virtual machine uses an optimised execution strategy in which functions defined within the program are compiled to native code, which can be executed much faster than individual manipulation of each graph node. This is essentially a hybrid approach which has similarities with normal execution of compiled sequential code, but allows the execution context to be quickly switched to a different part of the program whenever an I/O operation blocks. This enables parallelism to be achieved while still maintaining the performance benefits of compiled code execution.

Several existing functional language implementations use execution techniques that are very similar to that of NReduce, and these implementations have strongly influenced our design. The basic approach we have used is not new, but our implementation does have some important differences compared with existing work:

- Firstly, it supports automatic parallelisation, in contrast to other implementations, which require manual annotations. The reason why other parallel functional languages require annotations is that they target programs in which all of the computation is internal, and thus need to minimise the overheads associated with parallelism.
- Secondly, the I/O support in NReduce permits network connections to be accessed within pure expressions, on the assumption that the programmer is only accessing side effect free services. This contrasts with the imperative style of programming required in other functional languages whenever I/O is to be performed. Our approach enables multiple services to be invoked in parallel.
- Finally, we support an internal array-based representation of lists, which allows streams of binary data to be efficiently represented, without requiring explicit support for arrays in the programming model. Programmers can work with text and binary streams in the same manner as other types of lists, with efficiency gains possible when using high-level list manipulation functions which work for both representations.

It is hoped that the implementation techniques described in this chapter will be of use to other workflow engine developers who are looking to provide their users with a greater degree of expressiveness and efficiency. We believe that the evolution of workflow languages is likely to follow that of scripting languages, with more and more features being incorporated from general-purpose programming languages. If this trend continues, programmers will benefit from the ability to implement more of their application logic in a high-level workflow language in which parallelism is easy to achieve, instead of having to fall back to traditional imperative languages to implement the more complex parts of a workflow.

Chapter 5

XQuery and Web Services

The emergence of web services as a means of exposing application functionality over the Internet presents a rich set of opportunities. Applications can be constructed using standards-based protocols for communicating with services, facilitating the incorporation of functionality from multiple sources. Leveraging this functionality requires programming languages which provide developers with substantial flexibility as to how these services may be used, while simultaneously abstracting over low-level plumbing issues such as network protocols and data encoding. The widespread use of XML within the web service community suggests that a language which supports both native XML processing *and* access to web services could be of great value.

XQuery, discussed in Sections 2.6 and 3.8, is a functional programming language designed for manipulating XML data. The current language standard does not include support for web services, and existing implementations are primarily targeted at performing database queries. However, we believe that with appropriate extensions to enable service invocation, it is ideally suited for use as a workflow language in the context of the web services stack. In this chapter, we describe our implementation of the language, which includes support for web service access, and leverages the programming model and virtual machine described in Chapters 3 and 4.

We argue that XQuery, and in particular the way in which we have implemented it, demonstrates a compelling solution to the problem of providing a suitably expressive workflow language. Section 3.3 outlined several limitations of existing workflow languages with regards to language expressiveness. Although mainstream imperative and object oriented programming languages provide ample facilities for data manipulation and computation, they lack automatic parallelisation, integrated access to services, and native support for manipulating the XML data that web services send and receive. XQuery combines a range of computation and XML data manipulation capabilities with a functional programming model, making it ideal for use as workflow language.

This chapter describes our implementation of XQuery, which consists of a compiler that translates from XQuery to ELC, and a runtime library providing supporting functionality used by the generated code. Central to our implementation of the language is the `import` `service` statement, originally proposed by Onose and Simeon [243], with which the programmer can define mappings between namespace prefixes and web services. Based on these mappings, function calls which use the relevant namespace prefixes cause operations to be invoked on the corresponding web services. Calls to services are thus expressed in the same manner as regular function calls.

Our implementation is greatly simplified by the fact that all of the execution logic, including parallelism and handling of multiple network connections, is implemented by the NReduce virtual machine. This separation of concerns permits our XQuery compiler to operate purely in terms of language functionality, expressing XQuery language constructs in terms of ELC expressions which perform the equivalent computation.

5.1 Benefits of XQuery as a workflow language

Although it is intended mainly as a query language for databases and other sources of XML data, XQuery is in fact a Turing complete [307] programming language that can be used for more general types of programming. The computational abilities of the language are suitable for a wide range of data processing tasks. It is in this context that we consider it appropriate for use as a workflow language, since workflows are essentially programs which utilise externally-provided functions accessible as services.

The most important properties which make XQuery useful for expressing workflows are as follows:

- **Lack of side effects.** XQuery is a pure functional language, with no notion of state. This opens up the possibility of automatic parallelisation, which can be achieved by analysing the structure of a program to identify data dependencies, and using this information to determine which operations may safely run in parallel. As per the Church-Rosser theorem [69], the lack of side effects guarantees that the result of evaluation is independent of the order in which expressions are evaluated, provided that data dependencies respected. For workflows, we require the accessed services to also be free of side effects in order to guarantee deterministic results.

- **Native support for XML processing.** XQuery incorporates XML directly into its type system, and provides numerous high-level constructs for accessing, manipulating, and creating XML data structures. These constructs are much simpler to use than APIs such as DOM (Document Object Model) [142] which are provided by other languages. For workflows, XQuery permits data exchanged with web services to be manipulated directly, instead of going through a potentially lossy translation process [228]. It also removes the need for shim tasks [151] required by workflow languages that lack built-in data manipulation features, as discussed in Section 2.3.8.

- **Flexible control structures.** Support for control flow constructs is a crucial issue for workflow languages, which have traditionally been lacking in this area.

XQuery provides conditional branching, loops, recursion, user-defined functions, filter expressions, and tree traversal constructs which all fit cleanly within the purely data-oriented model of the language. The way in which many of these features operate opens up the possibility of data parallel execution, which is implicitly supported by our implementation due to the parallel evaluation mechanisms of the underlying virtual machine.

- **Concise syntax.** While XML is a good choice for encoding *data* exchanged between programs, it is a poor choice for *code*. Unlike many existing workflow languages, XQuery does not suffer from what we call *gratuitous XML syndrome*, a condition in which a language designer chooses XML for no apparent reason other than supposed ease of parsing. A concise syntax such as that provided by XQuery is vital for programmer productivity. The use of XML as a data model is an entirely separate issue to language syntax.

- **Standardisation.** XQuery is a mature, well-known language, standardised by the World Wide Web Consortium, and is implemented by many database systems and XML processing tools. Compared to designing an entirely new language, extending XQuery with workflow capabilities has the advantage that people who are already familiar with the language can reuse their existing skills, and people who are new to the language have a lot of good quality documentation available to them.

5.2 Language overview

XQuery is designed for performing queries on sources of XML data to extract elements matching specified criteria. In addition to the querying capabilities, it also includes the ability to create new XML content, and perform arbitrary computation. An XQuery program, often referred to as a *query*, consists of an (optionally empty) set of function definitions, followed by an expression which comprises the main part of the program. As is typical of functional languages, there is no such thing as a statement — all computation is performed by expressions, which can return a result, but not manipulate state. The outcome of executing an XQuery program is a sequence of items, which may subsequently be written to a data stream by the host environment.

5.2.1 Data types

XQuery supports two categories of data types: *nodes* and *atomic values*. The former may be any type of node that can be present within an XML document tree, including elements, text nodes, processing instructions, and comments. Atomic values include integers, floating point numbers, boolean values, and strings. Every kind of expression within an XQuery program is defined to return a *sequence* of items, where each item is an atomic value or node.

Because XML nodes are supported directly within the type system, and all structured data is represented as trees of nodes, there is a direct correspondence between what can be stored in an XML document and what can be represented at runtime by the XQuery program. This avoids the type mismatches that can occur when translating XML into objects for representation in an object-oriented language such as Java. Regarding serialisation, the only types which cannot be directly stored in an XML file are atomic values and sequences of multiple items. However, atomic values are implicitly converted into text nodes during serialisation, and sequences are included in a serialised document simply by concatenating the resulting serialised versions of each. These conversions are a standard part of XQuery, and are used when producing output.

5.2.2 Expressions and control flow constructs

Standard control flow constructs such as conditionals, loops, and recursive function calls are supported. There are no language constructs to manipulate state, so the only effect of a function call or expression evaluation is the computation of its result. Loops operate in a similar manner to `map` in functional programming languages; they evaluate the body for each item in a given sequence, and return a new sequence containing the results produced by each invocation of the body.

Expressions can include standard arithmetic, relational, and boolean operators, as well as several other forms specific to sequence processing, such as filtering elements based on predicates. The result of an expression may be used in any place that a regular value may be, such as input to a conditional test, `for` loop, or function parameter, as well as the result of the query as a whole. XQuery programs do not produce output explicitly, but instead return an item sequence which can then be output by the host environment.

5.2.3 Path expressions

XQuery includes the ability to select parts of an XML document based on *path expressions*. Each such expression is evaluated relative to a *context node*, which is a particular node in the input document, and the result of a path expression is the sequence of nodes that match the selection pattern.

Since all structured data in XQuery is represented a a tree of XML nodes, path expressions can be used as a means of accessing components of a data structure. The field access operator in object oriented languages, e.g. `->` in C++ or `.` in Java, roughly corresponds to the `/` operator in XPath, which enables a child of a particular element to be selected. The `/` operator is more general however, as it permits multiple children to be selected, supports other types of relationships (such as ancestor or sibling), and can be used in conjunction with predicates to filter the selection to only those nodes that match a particular condition.

Path expressions provide a very flexible way to access different parts of a data structure. By avoiding the need for explicitly-coded tree traversal logic, they enable many forms of tree access to be expressed in much less code than would be required in an imperative language. Within the context of workflows, they open up the possibility of very concise expression of processing that involves extraction of particular components of a web service result (possibly as a list of items), so that further computation and service calls can be made using each.

5.2.4 Element constructors

XQuery also includes the ability to construct trees of XML elements, using *element constructors*. These use the same syntax as regular XML, where the elements and attributes may be included as literals within the source code of an XQuery program. Additionally, parts of the tree, such as attribute values or certain branches, may be computed based on the results of further expressions.

This capability is useful for two main purposes. The first is the production of an output XML document, which contains data computed from the input. In this sense, XQuery can be used for performing transformations between different data formats, in a similar manner to XSLT [179]. Complex queries that perform computation (as opposed to just selecting nodes via path expressions) usually construct elements to contain the data they return.

Another use for element constructors is to produce data structures that are to be used within the query. Whereas in an object-oriented language, the programmer would define a class, and create instances of this during execution to represent objects, in XQuery the equivalent is to construct XML elements with child nodes or attributes representing the fields of the structure. In the context of web service access, this capability is useful for producing input values for web services that accept data structures, rather than atomic types, as parameters.

5.2.5 Example

Figure 5.1 shows an example of an XQuery program which uses all of the major language features discussed above. This program takes an input XML file containing an order for a set of products, structured as an `order` element with a series of `item` children, and produces an invoice, applying discounts and calculating the total value in the process.

The first part of the program produces a series of `line` elements, one for each `item` element in the original document. For each of these, it calculates the cost, based on the price and quantity of the item. Additionally, if the `item` element has a `type` attribute equal to “Special”, then a 20% discount is applied. This logic demonstrates the use of `for` loops, conditional statements, path expressions, and element construction.

Subsequently, the total value of all lines in the invoice is calculated. This is achieved by selecting all `amount` attributes of the previously-constructed `line` elements, and computing a sum of this sequence. The sum is assigned to the variable `total`, which is later used when producing the result. This logic demonstrates how data structures (element trees) created in one part of the program can be accessed from another part.

```
let
  $lines :=
    for $item in /order/item
    let $cost := $item/@price * $item/@quantity
    return
      if ($item/@type eq "Special") then
        <line name="{$item/@name}" amount="{$cost * 0.8}"/>
      else
        <line name="{$item/@name}" amount="{$cost}"/>,
  $total := sum($lines/@amount)
return
  <invoice total="{$total}">
    {$lines}
  </invoice>
```

Figure 5.1: XQuery example

Finally, the last part of the code produces the result document. This contains an `invoice` element, with a `total` attribute set to the previously computed total. Within the `invoice` element, the `lines` variable is used to include as children all of the `line` elements that were previously constructed. The result thus consists of a combination of data computed within the program, and a literal element specified directly in the code.

5.2.6 Relationship to other languages

XQuery is closely related to two other XML processing languages defined by the W3C. One of these is XPath (XML Path Language) [36], a notation for expressing the selection of a subset of nodes within a document. XPath uses a syntax similar to that of paths in a file system, where parts of the tree are specified in sequence, separated by slashes to indicate parent-child relationships. XPath supports a rich set of selection constructs which enable multiple nodes to be selected based on pattern matching and predicates. An example of an XPath query is `/book/chapter/@title`, which selects the `title` attributes of all `chapter` elements which are inside `book` elements, under the root of the document. XQuery is a superset of XPath.

Another closely related language is XSLT (eXtensible Style sheet Language Transformations) [179]. XSLT is designed for *transformations* from one XML format to another. It is also a pure functional programming language, uses the same data model as XQuery, and includes support for XPath expressions. The main differences between XQuery and XSLT are that the latter includes support for template-based pattern matching, and has a more verbose syntax in which control flow constructs are expressed as XML elements. We originally considered using XSLT as a workflow language [181, 183], but instead decided on XQuery, since it provides a more concise and convenient language syntax. Semantically however, both languages are very similar, and most of the ideas we discuss in this thesis relating to XQuery could also be applied to XSLT.

The language features we have described up until now are existing parts of the language supported by all implementations. In the following sections, we describe how we have extended the language with support for web services, and developed an implementation to demonstrate how XQuery can be compiled into ELC.

5.3 Web service support

The way in which we incorporate web service support into XQuery is designed to be as simple to use as possible. We achieve this by permitting service operations to be invoked using the regular function call mechanism already provided by the language, and allowing programmers to define mappings that determine which function calls correspond to web services. This is an approach similar to that taken by existing remote procedure call toolkits for imperative languages. Unlike these existing mechanisms however, our implementation achieves latency hiding by permitting multiple requests to be made in parallel, and avoids a potentially lossy conversion process by using the same data format for the language type system as is used by the on-the-wire format, i.e. XML.

The way in which we expose web services reflects the idea of treating the network as a platform, whereby a computing environment consists not just of a single machine with an operating system and libraries installed, but rather a set of computing resources available on a network which together provide a collection of useful functionality. When someone is writing a program, they should be able to focus more on *what* their program does, rather than *how* it does it. Just as high-level languages relieve the programmer from having to deal with low-level concerns such as the physical layout of memory, the programmer should not need to be concerned with the internal details of how a function is invoked over a network. Parameter marshaling, data transmission, and parallel execution should all be taken care of by the runtime system, instead of being exposed directly to the programmer.

5.3.1 The `import service` statement

In order to expose web service operations as functions, we need a way to map between a set of functions and a web service. In XQuery, a collection of functions is identified by an XML namespace, which in this context plays a similar role to a package name in Java. All function calls in XQuery are *namespace qualified*, meaning that they consist of a namespace prefix and a local name. The namespace prefix corresponds to a URI, which determines the “package” that the function call is in. By associating certain namespaces with web services, we are thus able to indicate which function calls correspond to web service operations.

To map a namespace to a web service, the programmer uses the `import service` statement [243], the syntax of which is as follows:

```
import service namespace <prefix> = "<url>";
```

The URL must correspond to the address of a WSDL file containing the interface definition for the service. For example, in order to associate the prefix `store` with the URL `http://store.com/apis?WSDL`, the programmer would write:

```
import service namespace store = "http://store.com/apis?WSDL";
```

Once this statement has been written at the top of the query, the `store` namespace prefix can be used anywhere within the query to make calls to this service. All that is required is that the programmer use the regular function call mechanism to specify the name of the operation and parameters, making sure that the namespace prefix `store` is used on the function call. For example, the following code returns the name of each item of clothing available in the store:

```
for $product in store:getProducts("Clothing")/*
return $product/@name
```

During compilation, the WSDL files referenced from `import service` statements are downloaded and analysed to determine all of the details necessary to communicate with the service. This process is described in Section 5.3.3.

5.3.2 Mapping function calls to SOAP requests

When a function corresponding to a web service operation is called, the operation is invoked by having the client send a request over the network, and subsequently receive the response. Although the parameters are already in XML, additional steps are needed to correctly build the request message containing the appropriate structure and to include the method name. Both the request and response are transmitted as SOAP messages over HTTP.

When the compiler encounters a service call, it translates this into a call to a stub function, which is generated as described in Section 5.3.3. The purpose of this stub function is to generate a SOAP request, send it to the service, and then parse the response to extract the result of the service operation. The interaction with the service is achieved using the built-in `connect` function, described in Section 3.5.8.

A function call of the form `calc:max(3,4)` would be serialised to the SOAP request shown in Figure 5.2. The `Envelope` and `Body` tags are standard parts of every SOAP message, and all data to do with the service call is contained within these. Directly inside the `Body` is an element whose name corresponds to the operation that is to be invoked. This element is in a namespace which is used by the service, and is not necessarily the same as the service's address. Within the operation element are child elements for each of the parameters passed to the operation. In this particular example, both parameters are atomic values, but XML structures can also be passed to web services within these elements as well.

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <msg:max xmlns:msg="urn:calc">
      <a>3</a>
      <b>4</b>
    </msg:max>
  </soap:Body>
</soap:Envelope>
```

Figure 5.2: SOAP request

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <msg:maxResponse xmlns:msg="urn:calc">
      <return>15</return>
    </msg:maxResponse>
  </soap:Body>
</soap:Envelope>
```

Figure 5.3: SOAP response

The response message, shown in Figure 5.3, has a similar structure. The `Body` has a child element whose name is formed by concatenating the operation name and `Response`. Within this is a `return` element containing the result of the operation. The names of these two inner elements depend on the SOAP implementation used on the server.

In order to determine the exact format of this message, it is necessary for the compiler to consult the WSDL file to find out how certain parts of the request should be expressed. This includes the parameter names of the operation (in this case, `x` and `y`), as well as the namespace to be used for the operation element (in this case, `urn:calc`).

Additionally, the compiler must also determine what message style the service uses, as this also affects the interaction. There are several different message styles used by web services [56], all of which achieve essentially the same effect. For simplicity, our XQuery implementation only supports the RPC/Literal message style. Support for other formats such as Document/Wrapped would be needed in a production system, but the minor syntactic differences between the styles are not relevant to the general concepts we explore here.

5.3.3 Stub functions

For each operation provided by services referenced from `import service` statements, the compiler generates a stub function which can be used to invoke the operation. These are

```
declare function g_0_max($a as xsd:int, $b as xsd:int) as xsd:int
{
  xq::fn_post("http://localhost:8080/calc",
    <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
      <soap:Body>
        <msg:max xmlns:msg="urn:calc">
          <a>{$a}</a>
          <b>{$b}</b>
        </msg:max>
      </soap:Body>
    </soap:Envelope>)/child::node()
                     /child::node()
                     /child::node()
                     /child::node()
                     /child::node()
};
```

Figure 5.4: Stub function for operation `int max(int a, int b)`

generated during an early stage of compilation before the XQuery code is transformed into ELC. The stub functions appear to the rest of the code as regular user-defined XQuery functions; the compiler is thus able to translate calls to these functions in the same way as calls to other types of functions. Figure 5.4 shows the stub function which would be generated for the `calc:max` operation discussed previously.

The name of the stub function is generated from the service URL and operation name. Since URLs may contain certain characters that are not permissible in ELC identifiers, an automatically generated prefix (in this case, `g_0`) is used instead. Calls to the function in the rest of the XQuery source code are transformed such that they reference the computed name rather than the original prefix:name combination. This process is also performed for local functions, since different prefixes can potentially refer to different namespaces in different parts of the code.

The stub function builds a SOAP envelope containing the message used to invoke the service. As discussed in Section 5.3.2, this consists of `Envelope` and `Body` elements, which in turn contain an element corresponding to the operation name, and a child element for each parameter. In the code shown in Figure 5.4, the `{` and `}` brackets are used to escape from literal text into XQuery expressions which contain references to the parameters supplied to the function. Thus, the parameters are included in the SOAP message verbatim.

Once the envelope has been built, it is passed to an ELC function called `xq::fn_post`. This produces a HTTP request containing the appropriate headers and the SOAP message, and passes this request to the server using the `connect` function, with the server address extracted from the supplied URL. The `xq::fn_post` function then extracts the content from the HTTP response, parses it into an XML tree, and returns it.

The XML data contained in the response is in the form of a SOAP envelope, as shown in

Figure 5.3. In order to obtain the actual result of the operation, it is necessary to drill down into the response tree to get to the element containing the result. The five `child::` steps traverse into the `Envelope`, `Body`, `calcResponse`, and `return` elements, then finally obtain the data inside the `return` element. This data is the result of the service call, which is returned to the caller.

5.3.4 Parallelism

No special consideration needs to be given to parallelism in the web service support of the XQuery compiler, since it is handled entirely within the NReduce virtual machine. The compiler simply needs to generate code which makes use of the built-in `connect` function to exchange data with remote servers, and the mechanisms described in Section 4.5 will take care of handling multiple concurrent requests. Since XQuery is a side effect free functional language, multiple service calls with no data dependencies between them can be safely executed in parallel, assuming that the services themselves are also side effect free.

Many language constructs in XQuery are able to take advantage of parallel execution. Iteration constructs such as path steps, filter expressions, and `for` loops can be executed in a data parallel manner, since each iteration is independent of the others. Binary operators such as `+` and `*` can have both of their operands evaluated in parallel. Function calls can have all of their arguments evaluated in parallel. When using our implementation, a programmer writing an XQuery program to access web services is able to exploit parallelism without including any explicit parallel directives in the code.

Parallel execution in our system relies on the services accessed within a workflow being able to handle multiple requests at a time, and execute them on different CPUs. For individual machines, this is often possible due to the presence of multi-core processors. For server farms, this is possible where a load balancer is in use which accepts requests at a single URL, and distributes those requests among different physical machines. The role of the workflow engine in such a case is not to provide the parallelism on the server(s) as such, but rather to make it easy for programmers to take advantage of it.

Cases in which multiple, identical services are hosted by different sites are not explicitly supported by our system, since we rely on the assumption that each service has a single, unique URL. However, it is possible to use services deployed in this manner using a client-side proxy which accepts requests sent to `localhost`, and redirects these requests to the services located at different URLs. This arrangement is effectively equivalent to a load balancer, and is what we use for our experiments in Chapter 7.

5.4 Usage examples

Having explained the support for accessing web services we have incorporated into XQuery, we shall now look at some examples of how it may be used. We shall begin by examining

```
import service namespace news = "http://reuters.com/api.wsdl";
import service namespace stocks = "http://nasdaq.com/stocks.wsdl";
import service namespace weather = "http://bom.gov.au/weather.wsdl";

<html>
  <head><title>Home page</title></head>
  <body>
    <h1>News:</h1>
    {news:getHeadlines('World')}
    <h1>Stocks:</h1>
    {stocks:topPerformers()}
    <h1>Weather:</h1>
    {weather:getWeather('Adelaide')}
  </body>
</html>
```

Figure 5.5: Web page generation incorporating different information sources

simple examples which simply retrieve data from existing services, and then look at a more complex example involving parallel evaluation of compute-intensive service calls, and internal computation based on the results of these calls.

5.4.1 Collating content from multiple sources

Many web pages incorporate data from several different sources. One example is a portal site showing news, weather, stock quotes, sports results, and textual advertising. Another example is a social networking site which shows a friends list, status updates, upcoming events, and message notifications. When a web server constructs such a page, it must send requests off to several different servers, and then collate the results together into a HTML page which is sent back to the client.

A straightforward implementation of such a web page in an imperative server-side scripting language would make these requests sequentially. The time taken to generate the page would be the sum of the individual request times, each of which involves network latency in addition to the time taken by the remote server to process the request. If the requests could instead be made in parallel, then the web page could be generated much faster, its time limited to that of the longest external request plus any local processing. Our implementation of XQuery makes this easy.

Figure 5.5 shows an example of an XQuery program which produces a HTML page containing data retrieved from several different web services[1]. This could be deployed on a web server to generate the front page of a site, resulting in faster response times due to

[1]All of the services used in these examples are fictitious, but are representative of the sort that would be useful in these contexts.

the fact that all of the service calls are made in parallel. The outer-level part of the script contains literal XML elements, which can be included verbatim in XQuery. Inside the sections bracketed with `{` and `}`, function calls are made to services to obtain data to be included verbatim in the page. This example assumes that the data is returned in HTML format; a more realistic implementation would likely include functions to transform the data from whatever format the service uses into HTML.

5.4.2 Processing of data structures returned from services

Some web services return structured data, such as records, lists, or trees. Most workflow languages treat the data returned by web services as opaque values, and provide no way to extract their components or traverse the data structures. XQuery provides many built-in language constructs for accessing XML content, which makes it easy to inspect structured data returned from web services. Additionally, element constructors can be used to create data structures which are accepted by web services as parameters.

Consider a workflow which reports the names and salaries of all employees in a company. The HR systems of the company provide a web service with two operations: `listEmployees`, which returns a list of employee IDs, and `employeeInfo`, which accepts an ID as a parameter and returns a record structure containing various information about that employee, such as the following:

```
<employee>
  <id>12960</id>
  <first-name>Bob</first-name>
  <last-name>Smith</last-name>
  <position>Technician</position>
  <department>Engineering</department>
  <salary>55000</salary>
  ...
</employee>
```

The goal of the workflow is to produce a list of elements containing the ID, full name, and salary of each employee, like this:

```
<employee id="12960" name="Bob Smith" salary="55000"/>
<employee id="12823" name="Gary Johnson" salary="65800"/>
<employee id="13012" name="Joe Williams" salary="49250"/>
...
```

In order to produce such output, it is necessary to perform two types of data structure access. The first is to access the list of strings returned by the `listEmployees` operation, and iterate over the items. Within each iteration, the `employeeInfo` operation must be invoked with the employee id, and the workflow must extract the `first-name`, `last-name`, and `salary` elements from the data structure passed back as the result. The first two of these must be concatenated together to form the full name, and then the combined

```
import service namespace hr = "http://localhost:8080/hr?WSDL";

<results>{
for $id in hr:listEmployees()/text()
let $info := hr:employeeInfo($id)
return <employee id="{$id}"
                 name="{$info/self::firstName,' ',$info/self::lastName}"
                 salary="{$info/self::salary}"/>
}</results>
```

Figure 5.6: Employee information workflow

full name, salary, and ID must be added as attributes to a newly-constructed `employee` element.

Figure 5.6 shows a workflow which does this. It uses a `for` loop to iterate over all of the items returned by the `listEmployees` operation. For each of these, it assigns the result of the `employeeInfo` call for that ID to a variable called `info`, and then uses path expressions to access the elements of the structure. A literal element constructor is used to create the `employee` element, with attribute values computed by accessing the `id` variable and the content of the employee structure.

5.4.3 Two-dimensional parameter sweep with graphical plotting

Task-farming systems such as those described in Section 2.1 are often used for *parameter sweep* applications, in which the same program is run multiple times, each with different parameter values selected from specified ranges. To perform a parameter sweep, the user specifies how many parameters each task takes, and the values over which each parameter can range. The middleware then executes one task for each combination of parameter values. This style of computation can be expressed in XQuery using `for` loops to express iteration over parameter ranges, with web service invocations corresponding to the tasks. All iterations of a `for` loop can potentially run in parallel.

As an example, consider an environmental monitoring application which tracks the rate of deforestation in a particular geographical area. A collection of satellite photos is obtained each year covering the area of interest, and these photos are stored in a database. A web service is provided which is capable of analysing regions within the images to determine the percentage of land covered by forest. By comparing this percentage over multiple years, it is possible to determine the rate at which forests have been cleared.

In order to produce a visual representation of the deforestation, the area is split up into sub-regions, arranged in a two dimensional grid. A parameter sweep is performed in which each task analyses a different sub-region, identified by a latitude, longitude, width, and height. Each sub-region is analysed over two different years, to determine the change in coverage. The results from this process are rendered into an isometric 3d plot. Using

```
import service namespace forestry = "http://environ.gov/forestry?WSDL";

declare variable $steps := 32;
declare variable $minlat := 5.26;
declare variable $maxlat := 8.82;
declare variable $minlong := 85.38;
declare variable $maxlong := 88.94;
declare variable $latmult := ($maxlat - $minlat) div $steps;
declare variable $longmult := ($maxlong - $minlong) div $steps;

declare variable $width := 1000;
declare variable $height := 400;
declare variable $cellwidth := $width div (2*$steps);
declare variable $cellheight := $height div (2*$steps);
```

Figure 5.7: First part of the parameter sweep workflow

XQuery, we can perform the parameter sweep as described above, and produce output in SVG (Scalable Vector Graphics) [1] format, which is based on XML. The built-in computation and data manipulation facilities of the language can be used to calculate the coordinates and surface shading in the plot based on the results obtained from the parameter sweep.

To implement this example, we shall make use of several constants, defined in Figure 5.7. These include the minimum and maximum latitude and longitude, as well as the number of steps in each dimension, which together determine the values over which the parameters will range. The size of the region to examine in each task (`latmult` and `longmult`) is calculated based on these values. We also specify the width and height of the output image, from which the size of each individual cell in the graph is computed. At the beginning of the file we import the forestry service, which is used to perform the image analysis. Since there are 32 steps in each dimension, and two service calls for each area (to analyse images from two different years), there will be $32 \times 32 \times 2 = 2{,}048$ tasks invoked by this workflow.

Figure 5.8 shows the main part of the workflow. Lines 8 – 20 perform the parameter sweep, using `for` loops to iterate over each dimension. There is one iteration for each step in each dimension, in which the appropriate values for latitude and longitude are calculated based on the minimum and maximum values specified earlier, and the current step. The outer loop produces a sequence of `row` elements, and the inner loop produces a sequence of `col` elements. Each `col` element contains the difference in percentage of forest coverage between the years 2004 and 2009. This is computed by calling the `analyse` service operation once for each year, and subtracting the results. The parameter sweep executes in a data parallel fashion, since each loop iteration is independent of the others, and all of them are sparked immediately.

Lines 23 – 38 produce an SVG document which displays the results in an isometric 3d

```
declare function local:position($row,$col,$value)
{
  ($cellwidth*(1 - $row + $col) + $width div 2,
  ",",
  $cellheight*($row + $col - 1) - floor($value))
};

let $data :=
  for $y in 1 to $steps
  let $lat := $minlat + $y * $latmult
  return
  <row lat="{$lat}">{
    for $x in 1 to $steps
    let $long := $minlong + $x * $longmult
    return
    <col long="{$long}">{
      forestry:analyse($lat,$long,$latmult,$longmult,2004) -
      forestry:analyse($lat,$long,$latmult,$longmult,2009)
    }</col>
  }</row>

return
<svg version="1.1" xmlns="http://www.w3.org/2000/svg">{
for $diag in 1 to $steps * 2 - 1,
    $horiz in max((2,$diag+1-$steps)) to min(($steps,$diag - 1))
let $row := $diag + 1 - $horiz,
    $col := $horiz,
    $colour := floor(155+$data[$row]/*[$col])
return
<polygon
  points="
  {local:position($row - 1,$col - 1,$data[$row - 1]/*[$col - 1])}
  {local:position($row - 1,$col,    $data[$row - 1]/*[$col])}
  {local:position($row,    $col,    $data[$row]    /*[$col])}
  {local:position($row,    $col - 1,$data[$row]    /*[$col - 1])}"
  style="fill:rgb({$colour},{$colour},{$colour});
         stroke:black;stroke-width:0.5"/>
}</svg>
```

Figure 5.8: Second part of the parameter sweep workflow

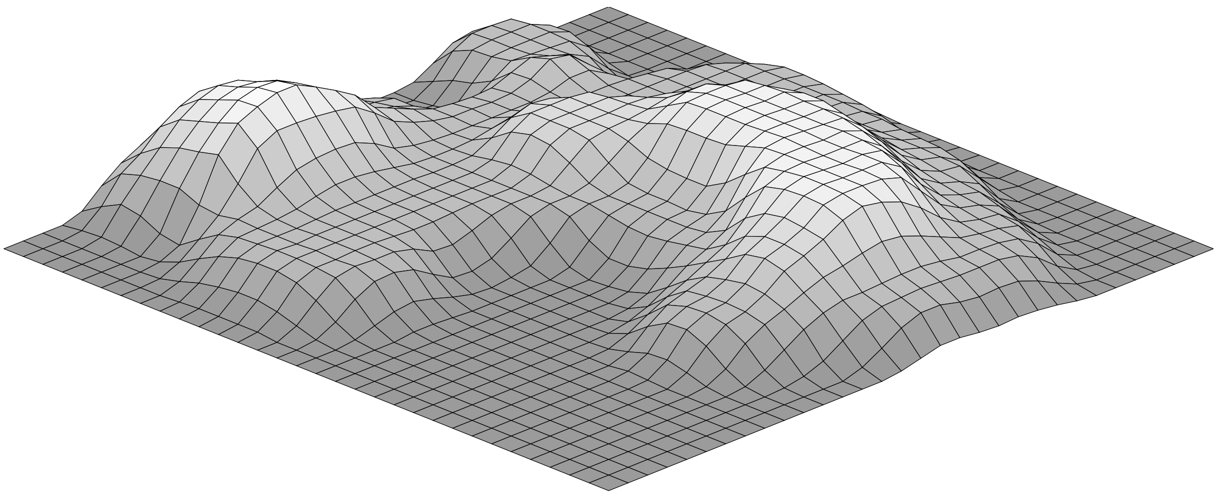


Figure 5.9: SVG file produced by the parameter sweep workflow

plot. In order to ensure the surfaces overlap in the correct manner, those which appear closer to the viewer are drawn after those which appear further away. This ordering is achieved by drawing the surfaces in the order of the diagonal on which they appear; the `for` loops on lines 24 and 25 iterate over each diagonal and each of the cells on that diagonal. Within the loop body, the current row and column are calculated, as well as the colour of the current cell. Colour components in SVG range from 0 – 255, so by adding 155 to the percentage value at the current point in the grid, we obtain a colour in the range 155 – 255.

On lines 30 – 37, a `polygon` element is generated for the current cell, with the height of the bottom-right corner taken from the value of the current cell, and the heights of the other corners taken from the values of its neighbours. The 2d positions at which these points lie are calculated using the `position` function on lines 1 – 6, which transforms a row and column number into an `x,y` coordinate which is included in the `points` attribute of the element. The element is shaded in grayscale using the previously-computed `colour` value.

Figure 5.9 shows an example of the output produced by this rendering process. This example is based on synthetic data — it was generated by running the workflow against a Java-based web service which simply returned the appropriate data points from a manually-generated data file.

This example illustrates how XQuery can be used for performing non-trivial calculations directly within the workflow, in addition to exploiting parallelism inherent in the structure of the code. The coordination and computation aspects are specified in the same notation, and the programmer is able to perform any manipulation they want on the result data returned by the service calls.

5.5 Implementation details

Our XQuery implementation uses ELC as a target language. This enables us to focus on the higher-level concerns of expressing the various language constructs of XQuery in terms of ELC expressions, and rely on NReduce to address the lower-level details such as memory management, native code generation, and parallel execution. In particular, the fact that NReduce automatically detects opportunities for parallel evaluation within ELC expressions means that as far as the XQuery implementation is concerned, automatic parallelisation comes "for free". Our compiler preserves the explicit data dependencies inherent in the structure of the source code during the translation. Since XQuery is a side effect free functional language, these dependencies are sufficient to determine which expressions may safely be evaluated in parallel.

Our use of ELC as a compilation target is intended to demonstrate how NReduce can be used as a platform for implementing higher-level workflow languages. The separation of concerns between the XQuery implementation and the virtual machine simplifies the design of both, and makes it possible to reuse the virtual machine for other workflow languages as well. For example, it would be possible to translate some of the graph-based workflow languages described in Section 2.3 into ELC, and then use NReduce as an execution mechanism for these languages. The use of a lambda calculus variant as an intermediate form is common in compilers for functional languages [161, 17, 163], but has not previously been used in the implementation of workflow languages. Additionally, if fault tolerance were to later be implemented, it could be done at the level of NReduce rather than the XQuery implementation. Specifically, retrying or rescheduling service requests could be handled within the data-oriented connection abstraction provided by the virtual machine. This fault tolerance support would then extend to other workflow languages implemented on top of NReduce as well.

In this section, we discuss several important aspects of how XQuery language semantics are mapped into ELC. We begin by describing the runtime library, a collection of ELC routines which provide support functionality needed by the generated code. Next, we explain how item sequences in the XQuery data model [101] are represented using cons lists. The *static context* and *dynamic context* of an expression, which correspond to information available about a given expression at compile time and runtime, are mapped to in-scope variables which may be accessed from within the code where necessary. Tree construction also needs to be specially addressed, since this involves certain complexities in the context of a strict functional language. Finally, we discuss the operation of the compiler.

5.5.1 Runtime library

The ELC code generated during compilation relies on a set of support routines which implement certain functionality required by XQuery. These are written in ELC themselves, as separate modules which are incorporated into the generated code using the `import` statement described in Section 3.5.2. A module import statement in ELC acts like an include directive, whereby the contents of the file are incorporated into the referencing file, and all of the code is compiled as a single unit. This simple mechanism is sufficient for the relatively small size of the runtime library, though as future work it may be useful to incorporate support for separate compilation into NReduce.

The reason for implementing all of the modules in ELC is so that they can take advantage of the parallel execution and networking features of the virtual machine. We initially considered implementing some of the functionality in C, but decided instead that implementing it in ELC would provide a greater separation of concerns between the virtual machine and XQuery implementation, and also enable certain functions which process lists of items to take advantage of parallel execution. This approach also means that the networking support within the virtual machine does not need to concern itself with application-level protocol issues such as HTTP request and response processing.

One of the modules is dedicated to XML processing. This contains a parser, which takes a data stream as input and returns a tree of XML nodes using the representation described in Section 5.5.2. It also includes routines for serialising these trees back into text format, which is used when generating SOAP requests and printing the final output of the program. Although the serialisation part was straightforward to implement, the parser implementation was complicated somewhat by the side effect free nature of ELC, which made it more difficult for parsing functions to keep track of their current position in the input stream while also returning syntax tree nodes. The technique used in this case is for each parser function to return a pair consisting of a node and the remaining portion of the input stream.

Numerous language constructs in XQuery require processing that is best encapsulated in a runtime support function, to avoid duplication in the generated code. For example, some comparison operators work on sequences, comparing every item in the first sequence with every item in the second until a matching pair is found. This involves an algorithm that uses nested loops to traverse the two sequences. Since this algorithm is identical for all uses of these operators, it makes sense to implement it in a library function, instead of repeating it every time such an operator is used. Avoiding duplication reduces the size of the generated ELC code, and hence the time required for bytecode compilation within the virtual machine.

The runtime library also provides implementations of XQuery's built-in functions and operators, specified in [214]. These are implemented in terms of other data structure manipulation operations provided by the library, and also the primitive operations of ELC. Most XQuery operators do not translate directly to primitive ELC operations, because of the need to perform type checking and handle sequences of items. For example, the addition operator must first check that both arguments are sequences of a single item, and that both items are numbers. It then extracts the values of the items, adds them, and constructs a new item tagged with the *number* type, and places it in a singular list. This type checking and list manipulation logic is implemented using operations provided by ELC.

The HTTP module builds on the networking functionality described in Sections 3.5.8 and 4.5. It constructs a request using ELC's string manipulation functions to append the headers and request body, and then passes it to the `connect` function, which then returns the response from the server. This response must be parsed to check if the request was successful, and to extract the body of the response. When used for accessing web services, the request and response are also passed through the XML serialisation and parsing logic. Basic HTTP features like redirects and chunked transfer encoding are also supported.

5.5.2 Data representation

XQuery's type system is based on the concept of *item sequences*. Every type of expression is defined to return a sequence, rather than an individual value. Sequences may contain any number of items, including zero. There is no provision to represent a single item by itself; whenever a single item is returned by an expression, it exists within a sequence containing just that one item. Sequences may not be nested; whenever two or more sequences need to be combined together in an expression result, the contents of each are concatenated together. Within the context of the workflow, each individual item is either an atomic value (such as an integer or string), or a node in an XML tree (such as an element, attribute, or text node). The full set of item types supported by our implementation is shown in Figure 5.10.

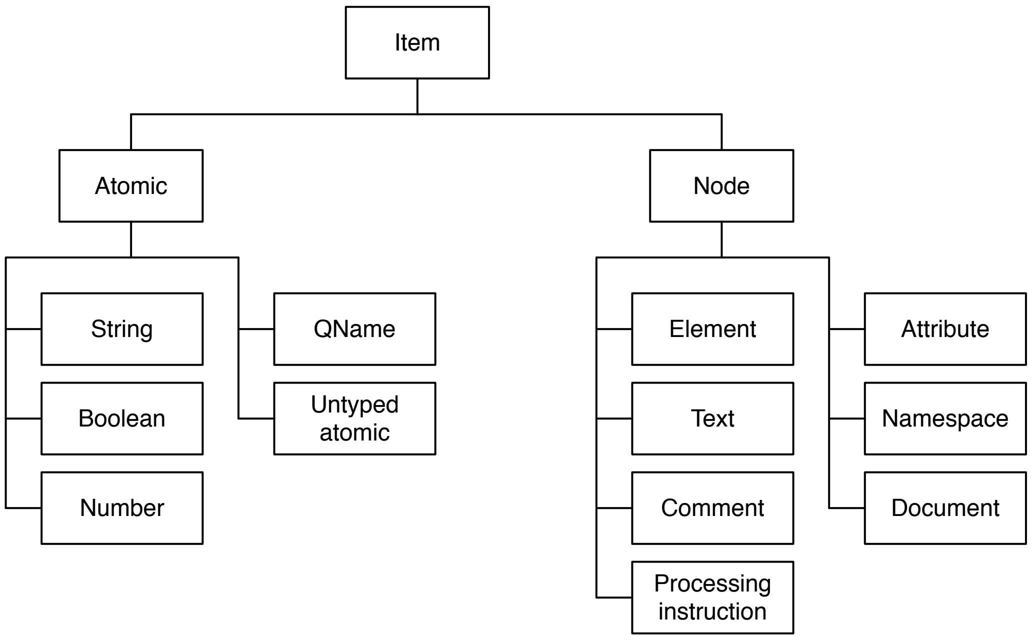


Figure 5.10: Types in the XQuery data model

Conceptually, an item can be thought of as an object of a particular class. In an object-oriented language, one would define a class for each type of atomic value or node, and every item would be an instance of one of these classes. However, as ELC follows a minimalist design, it has no notion of classes, and supports only four elementary data types: number, cons pair, nil, and function. To represent items, we must therefore use an abstraction which enables us to represent objects/records in terms of these data types.

This is achieved by representing each object as a list, with the first item containing a numeric type identifier, and the remaining items containing the fields of the object. This approach to representing structured data is based on that described in [2].

The XML support module contains a set of ELC functions to create different types of items. Each constructor function takes one parameter for each field, and returns a list containing the appropriate type identifier and field values. Inspecting items to determine their type is achieved by looking at the first element of the list, and accessing fields is achieved by retrieving the values from the appropriate positions within the list. Accessor functions are provided for each type of item and field, to make the generated code clearer and provide a sense of abstraction over the physical representation. While this approach is somewhat less efficient than would be possible with a target language that provided native support for objects, it is adequate for our purposes, and avoids complicating the design and implementation of ELC.

Item sequences are represented as lists. Therefore, every generated ELC expression corresponding to an XQuery expression returns a list. Whenever there is a need to perform processing on a single item within a sequence, the length of the list is checked, and then the item is retrieved using the `head` function. When two sequences need to be combined together, the `++` function is used to concatenate them, producing a new sequence containing all items from the original sequences. For iteration constructs such as `for` loops and path steps, a variant of `map` is used which concatenates the lists returned by each iteration, rather than placing them in a nested list. Although ELC permits lists to be nested arbitrarily, this flattening is necessary to comply with XQuery's requirement that sequences may not be nested.

5.5.3 Static context

The *static context* of an XQuery expression is the information that is known about that expression at compile time. It consists of several components, including the set of in-scope functions, variables, and namespace bindings. The scope rules of static context components follow the standard convention in which child nodes in the syntax tree inherit components defined in their parents, unless overridden locally by some language construct. For example, a variable bound by a `let` expression is in scope for the body of that `let` expression, except for those parts of the body that define a new variable with the same name, bound to a different expression. Some components of the static context, such as the set of available functions, are fixed for all parts of a query. Others, such as namespace and variable bindings, can sometimes be present in only parts of the query.

User-defined functions, which in XQuery can only be defined at a global level, correspond to function definitions in ELC. Each function in the generated code is given the same name as used in the original query, with the exception that namespace-qualified function names are given a name dependent upon their namespace URI, but not the prefix. All references to a given function use the simplified name, since they may potentially use different prefixes within the query. Global variables specified with the `declare variable` statement are handled similarly, as are variables bound by `let` and `for` expressions.

Namespace bindings are represented in the generated ELC code by variable bindings of the form `ns_P = "U"`, where `P` is the prefix and `U` is the URI. Global namespace declarations specified with `declare namespace` cause a top-level constant definition to be included in the generated code. Namespaces bound within literal element constructors are translated into letrec-bound variables. Wherever a namespace prefix is used in a query (as part of a qualified name), the generated code simply contains a reference to the appropriately-named variable. Since the scope rules of namespaces in XML and XQuery mirror those of variables in ELC, this approach implicitly relies on the symbol table logic within the ELC compiler to ensure that all references are mapped to the correct value.

In certain cases, such as when elements are constructed using names computed at runtime, namespace resolution cannot be performed statically. To support dynamic resolution, a list of (prefix,URI) pairs is bound to a variable called `namespaces`, which is in scope at all points in the syntax tree. At the top level, this contains only those namespaces which are globally defined. For each local namespace binding, found within element constructor expressions, the `namespaces` variable is re-defined to contain both the local namespace bindings and those present above that point in the tree. Namespace resolution is performed at runtime by passing the `namespaces` list and the computed prefix to a library function which searches through the list for a matching binding, and returns the URI. Since this dynamic resolution incurs runtime costs, it is only used where the prefix cannot be determined statically.

The XQuery specification defines several other static context components, which dictate the semantics of various language constructs. These components include the XPath 1.0 compatibility mode, construction/ordering mode, boundary-space policy, and default collation. In XQuery processors which implement the full specification, these settings can be overridden in (parts of) a query, giving the programmer control over how certain types of processing are performed. However, as the ability to change these options is not relevant to our research, we implicitly support only the default values. The static context components relating to types are also omitted, since our prototype does not support static typing.

5.5.4 Dynamic context

The runtime information maintained during evaluation of an expression in XQuery is referred to as the *dynamic context*. It consists of global information about the evaluation environment, the values of all local variables, and the current *focus* of the expression. The latter is used in certain types of iteration constructs to refer to the current item being processed in a sequence.

Local variables and function parameters in XQuery correspond directly to local variables and parameters in ELC, so the translation of these is straightforward. As in most programming languages, each function call maintains an *activation record*, or *frame*, in which the local data specific to that particular function call is stored; this permits functions to be recursive. In our implementation, it also allows multiple instances of an expression, such as different iterations of a loop, to be under evaluation at the same time. The specific details of how local variables are physically stored within a frame are handled by NReduce, so these do not need to be considered by our XQuery compiler.

The focus of an expression consists of three parts: the *context item*, *context position*, and *context size*. These are used when evaluating two specific types of iteration constructs: *path steps* and *filter expressions*. A path step of the form E_1/E_2 causes E_2 to be evaluated once for every item in E_1, with each evaluation of E_2 having the focus on one specific item. The result of a path expression is the concatenation of all the sequences returned by the iterations of E_2. Filter expressions of the form $E_1\,[E_2]$ are evaluated similarly, except that the result is a sequence containing only those items from E_1 for which E_2 evaluates to `true`.

In a sequential implementation, it is possible to store the context item, position, and size in global variables, pushing and popping them from a stack each time evaluation enters or exits an iteration construct. However, in our implementation, it is possible for there to be multiple expressions under evaluation at the same time, so we must instead maintain this information in a way that does not rely on there being a notion of a single, "current" iteration.

For both constructs, the code generator transforms E_2 into a lambda abstraction which takes the context item, position, and size as parameters, and is applied to the items in E_1 using a variant of `map` or `filter`. The variables `citem`, `cpos`, and `csize` are in scope within the lambda abstraction, so the components can be retrieved within the expression simply by referencing the appropriate variable. E_2 may in turn contain other path steps or filter expressions, in which case additional lambda abstractions of the same form will be generated. Whenever the compiler encounters a context item expression (`.`), or a call to the built-in functions `position()` or `last()`, it generates a variable reference to `citem`, `cpos`, or `csize`, respectively. As per the scoping rules of lambda calculus, these references will always correspond to the most closely-bound instances of these variables. The transformation rules for path steps and filter expressions are given in Section B.2.

The global information that the XQuery specification describes as being part of the dynamic context does not correspond directly to data values maintained at runtime, but rather external properties of the environment in which the query is being evaluated. Some of these pieces of information are only relevant to features not supported by our implementation, such as the current date and time, the local time zone, and the available collections. The only two properties of relevance in our case are the set of available function implementations and available documents. As far as our implementation is concerned, the function implementations are technically a static property, since they never change during execution. The set of available documents consists simply of all possible documents can that can be retrieved via HTTP requests; this does not require any explicit representation at runtime.

5.5.5 Tree construction

The simplest way to represent a tree in a side effect free language is to maintain references in only a single direction — from parents to children. The tree can thus be constructed in a bottom-up manner, with child nodes created before their parents. However, XQuery requires that references go both upwards and downwards in the tree, as well as sideways to sibling nodes; this is necessary to support axes such as `parent`, `ancestor`, `following-sibling`, and `preceding-sibling`.

If side effects were permitted, then it would be possible to simply create child nodes first, then parents, and later modify the child nodes so that they reference their parents. However, since our entire approach to parallel execution fundamentally relies on the lack of side effects, these modifications cannot be permitted. Additionally, strict evaluation provides no way to establish cyclic references between objects, because any given object must be initialised completely before a reference to it can be obtained. In a sense, this is like the "chicken and the egg" problem — one object cannot be created without the other already existing.

In order to support cyclic references, we use lazy evaluation for the construction of XML trees. Whenever a node is created, the constructor function is supplied with references to the already-existing parent and previous sibling nodes (if any). The following sibling and list of children are passed in as suspended evaluations [110], representing expressions that will later be evaluated to construct these nodes. When the children and following sibling are finally created, these references will then point to the correct objects. In order to achieve this when strict evaluation is being used by default, we use the laziness annotations described in Section 3.5.6 for the appropriate parameters of the node constructor functions.

This technique is only used in situations where there is a possibility that the references to parent nodes or siblings could be needed by an expression. When a tree is being constructed for direct output, the parent, previous, and next references are simply set to nil, because they are not used by the output routine. When the tree is assigned to a variable which could later be used in a path expression, a copy of the tree is made in which all the references are set, so that axes like `parent` and `following-sibling` will work correctly.

5.5.6 Compilation process

The full implementation details of the compiler are outside the scope of this thesis. Here, we describe the main stages involved with its operation:

1. **Stub generation.** For each `import service` statement, the compiler downloads and analyses the associated WSDL file to determine the operations provided by the service. For each operation, a stub function is generated, as discussed in Section 5.3.3. The stub functions appear exactly like regular user-defined functions to the

rest of the compiler. The compiler assumes that all operations exposed by services are side effect free.

2. **Namespace resolution.** All qualified function and variable names of the form prefix:localname are converted into unique, unqualified names. This simplifies the remaining parts of the compiler by avoiding the need to cater for different mappings from prefixes to namespaces throughout the code.

3. **Query simplification.** Certain constructs in the source code are translated into simpler, but equivalent forms. This simplifies the code generation stage by reducing the number of cases it has to consider. For example, a `where` clause associated with a `for` loop can be translated into an `if` expression within the body of the loop. This avoids the need for an explicit code generation rule for `where` clauses, enabling the rule for `if` expressions to be used instead.

4. **Code generation.** This is the main part of the compiler. It involves traversing the source tree and translating each XQuery expression into an equivalent ELC expression which computes the appropriate result. In some cases, the generated ELC code includes calls to library functions to assist with its operation. For example, path expressions are compiled into calls to a variant of `map`, where the supplied function is a lambda abstraction taking the context item, position, and size as parameters.

5. **Optimisation.** Certain simple optimisations are performed on the ELC code to eliminate operations that can easily be identified as redundant. This can occur due to the direct nature of the code generation phase, which translates each expression individually without considering the relationships to its children. For example, an expression may be generated which constructs a boolean value, and then immediately tries to convert it to a boolean value. Since the conversion has no effect, it can be skipped.

6. **Code addition.** This involves addition of variables representing top-level namespace declarations specified in the XQuery code, as well as a `main` function, which invokes the query. The job of the `main` function is to perform a small amount of initialisation, parse an optionally-specified input file, invoke the query, and then serialise its result to output.

Appendix B discusses these stages, particularly code generation, in more detail.

5.6 Discussion

Having described how our implementation works, we shall now discuss some of the lessons learnt during its development, as well as the benefits achieved by implementing XQuery on top of NReduce.

5.6.1 Factors that made implementation easy

An important factor contributing to the ease of implementation was the alignment between the programming models of XQuery and ELC. Both are side effect free functional languages, expressed purely in terms of functions and expressions that produce a result without manipulating state. The translation between the two languages was largely an exercise in expressing the semantics of XQuery in terms of lambda calculus. Although the translations were in some cases quite involved, the definition of support functions in the runtime library enabled the generated code to be kept fairly simple by delegating to these functions where appropriate.

The separation of concerns between the XQuery implementation and NReduce was vital in making the construction of both a practical endeavour. Intermixing the complexity of XQuery with the evaluation, parallelism, networking, and distribution mechanisms of the virtual machine would have been a nightmare; we believe that following this approach would have been very unlikely to have produced a successful result. NReduce is able to focus solely on efficiently executing ELC code, and the XQuery compiler and runtime library can concern themselves only with high-level language issues instead of execution details. ELC acts as a very clean and simple interface between the two layers. Since NReduce is completely independent of XQuery, it can also be used as a mechanism for executing other workflow languages that can be compiled into ELC.

Our choice of lambda calculus as the basic programming model meant that we had an intermediate language that was capable of expressing arbitrary computation, and was thus able to handle all of the implemented XQuery constructs. We had originally considered a dataflow model [182], but found that established techniques for expressing control structures like conditional branching, iteration, recursion, and list processing within functional languages offered what we considered to be a cleaner and more natural approach than their equivalents in dataflow languages. The widespread acceptance of lambda calculus as the fundamental model of functional programming also helped to convince us that this was an appropriate model to use.

The automatic parallelisation provided by NReduce was important for our XQuery implementation, since XQuery does not include any language support that enables the programmer to annotate expressions as candidates for parallel evaluation. Most existing functional languages require these annotations to be specified, in order to avoid the generation of large numbers of sparks which can affect execution efficiency. Although our approach of sparking every possible expression does introduce overheads, the fact that workflows delegate most their computation to external services implies that in this case, a degree of inefficiency of local computation is tolerable if it makes exploitation of parallelism easier overall.

From an engineering perspective, we found the use of the program-transformation language *Stratego* [49] extremely helpful in constructing our compiler. The ability it provides to define a set of high-level, formal rules specifying the transformation between source and target languages proved much simpler than previous experiences we have had with implementing compilers in C, including the bytecode compiler in NReduce.

5.6.2 Factors that made implementation difficult

XQuery is a very complex language. Implementing the full standard would have required a much greater investment of time than we deemed necessary to demonstrate our core research ideas, and would have provided nothing in the way of research benefit. For this reason, we chose to implement only a subset of the language, focusing on the most important language constructs. Among the features we omitted were static typing, XML validation, `union` and `intersect` operators, date support, and `order by` clauses in `for` loops. Many of the built-in functions and operators of XQuery are also not supported. Adding these features would largely be a matter of additional coding effort.

For some language constructs, we implemented versions with simpler semantics to those given in the specification. In some cases this was to simplify the implementation, and in others it was to avoid expensive overheads that with the original semantics, can only be avoided with sophisticated optimisation that would have involved yet more coding effort. An example of such a case is the specification's requirement that all results from path expressions are returned in document order, without duplicates. A naïve implementation of this would perform a sort for all path expressions, regardless of whether or not the nodes were already in document order. In the majority of cases this is already the case, and a sufficiently good optimiser would notice this and avoid the sorting unless absolutely necessary.

Support for axes such as `parent`, `previous-sibling`, and `following-sibling` also introduced difficulties in terms of tree construction, as explained in Section 5.5.5. In strict functional languages, it is not possible to create cyclic data structures, since doing so requires the ability to have a reference to an object before it is created. As discussed in Section 6.1, we default to strict evaluation for performance reasons. However, to permit the creation of trees with pointers in all directions, we needed to support laziness annotations, to allow the parameters to node constructors to accept suspended evaluations.

5.6.3 Performance implications

Certain choices we made regarding the in-memory representation of data structures were based on simplicity and ease of implementation, rather than efficiency. As a result, many operations must be performed at runtime to access these data structures that would not be required if a more efficient representation had been chosen. This impacts the compute-intensive performance of XQuery code and the scalability of the language implementation in terms of the sizes of documents it can handle. Primarily these decisions were made due to the lack of static typing both at the level of ELC and of XQuery.

Perhaps the most significant of these choices is the representation of objects using cons lists, as discussed in Section 5.5.2. We deliberately omitted from ELC any notions of static typing or structured types, with the exception of cons pairs. As a result, any time there is a need to store a collection of fields corresponding to an object, a list must be used in which each field is stored in a different position. Although these lists are stored as arrays at runtime, it is necessary to perform at least three operations every time a field is accessed: checking if the object is a list, verifying that the specified index is within the bounds of the list, and finally looking up the value at that index. This is slower than simply accessing a value at a particular offset relative to a pointer, as is typically done in statically typed languages like C++, Java, and Haskell.

Static typing of the XQuery code itself is also omitted from our implementation, which means that a large number of type checks have to be performed at runtime within the generated code and runtime library. Each item (either an atomic value or XML node) has a type tag associated with it, which is checked whenever there is a need to verify the type of an item, such as when performing an arithmetic operation. If static typing were supported, these checks could be skipped at runtime, which would likely improve performance significantly. In many cases, this would also permit values to be stored in unboxed format (without a tag), resulting in fewer memory allocations, and avoiding the need to package up and extract values from their tagged representations.

As with the missing features mentioned in Section 5.6.2, these performance issues could be alleviated through additional engineering effort invested into the XQuery compiler and runtime library, as well as NReduce itself.

5.6.4 Advantages over existing RPC mechanisms

Toolkits that implement remote procedure call functionality, such as those used for web services, expose remote operations as function calls. This enables distributed programs to be written using the same syntax and similar semantics to local programs, based on structuring the program and functionality that it uses into functions. This is a technique that is familiar to many programmers. The approach we have taken to exposing web services to XQuery programs follows this model. However, it does so in a way that avoids two major drawbacks associated with traditional RPC implementations.

In imperative languages, one of the main problems with exposing remote procedure calls using the same language construct as local calls is that remote calls involve significant latency. A programmer who writes code ignoring this latency may end up with a program that spends most of its time sitting idle waiting for other computers to do things, and not be able to carry on with other work or initiate other service calls in the meantime. The parallelism support of our virtual machine avoids this problem by permitting multiple function calls to be active at the same time, and preventing the blocking of one call from affecting others.

Another common problem with RPC toolkits arises from the fact that they perform *marshaling* whenever they invoke a remote call. This process serialises data structures from the programming language's own representation to an on-the-wire format that can be both transmitted as a stream of bytes, and interpreted meaningfully by the other end. In almost all cases, there is some degree of mismatch between what can be represented in the language's type system and the serialisation format [228]. This means that some of the data and objects a program manipulates cannot be exchanged with a remote service, and the language on the other end may not be capable of properly representing all information contained in the serialised format. In the case of an XQuery program accessing a web service, this is a non-issue (on the client side, at least), because XML is both the serialisation format *and* the language's native type system.

A third problem with remote procedure calls is that they can fail, due to network errors or crashes on remote machines. Although we do not address this issue in our implementation, the side effect free model of computation opens up the possibility of implementing fault tolerance by simply retrying failed service operations. Assuming the services meet the requirement of lacking side effects, this can be done safely. Incorporating retry and service substitution mechanisms into our implementation along the lines of those supported by Taverna [242] and Kepler [211][2] would greatly enhance reliability in the case of long-running workflows, and is an area we consider worthy of further investigation, as discussed in Section 8.4.3.

5.6.5 Advantages over existing workflow languages

Section 2.3.8 discussed several limitations of existing workflow languages which are alleviated by XQuery. Transformations between different data formats used by syntactically incompatible services can be performed using path expressions and element constructors to extract and combine data. Small amounts of application logic such as computations on values can be expressed using the expression syntax, utilising standard arithmetic, sequence, and string operations, as well as (potentially recursive) user-defined functions. Control flow constructs which are common in general-purpose programming languages but often missing from workflow languages are also supported by XQuery, making it possible to define complex workflows involving various forms of conditional processing, iteration, and data structure traversal.

Users with little programming experience are likely to find XQuery harder to understand than existing graphical workflow languages. Although it is tempting to blame this on the use of textual syntax instead of a visual notation, we believe that the (perceived) ease of understanding of existing workflow systems is primarily due to their simple but restrictive programming models. Regardless of what syntax one is working with, it is the *semantics* of the language that ultimately determines both understanding and capability.

Ease of understanding and ease of use are not the same thing, however. The lack of flexibility of existing workflow languages actually results in some things being *harder* to achieve, since whenever something needs to be done which cannot be expressed in the workflow language, the user is forced to add a shim task [151] containing custom code written in an embedded scripting language. These languages require a similar level of programming expertise to XQuery. Our philosophy is that a workflow language should be powerful enough to support the operations that are normally implemented in shim tasks, by supporting multiple levels of abstraction and a suitably general programming model.

Our choice of XQuery stems from the desire for a highly expressive workflow language which can easily manipulate data in the format used by web services, while also being

[2]The retry mechanisms supported by these systems inherently rely on the assumption that services are side effect free, since retrying a service with side effects could lead to incorrect behaviour.

based on a programming model that is feasible to automatically parallelise. In retrospect, we were quite lucky that XQuery already existed, as developing such a language from scratch would have been a major project in itself. XQuery is already a well-established standard, so users familiar with it can re-use their expertise with the language to develop workflows, and those that are new to the language have many learning resources available to them.

5.7 Summary

XQuery has many properties which make it attractive for use as a workflow language, in particular its native support for XML processing, and the fact that it is a pure functional language from which parallelism can be extracted automatically. We implemented an extension to the language which makes it possible to access web services from within XQuery, which enables it to be used for workflows.

In comparison with the other languages discussed in Section 2.3, XQuery makes it much easier to develop complex workflows. This is because it includes facilities for performing computation and data structure manipulation using the values exchanged between web services, avoiding the need for external shim tasks written in other languages. It also contains a range of flexible control flow constructs, many of which enable data parallel processing to be easily expressed.

The programming model introduced in Chapter 3, ELC, is designed as a generic programming language that can be used as an intermediate form by compilers of other workflow languages. We made use of this in our XQuery implementation, by developing a compiler which translates XQuery programs into ELC, as well as a collection of library routines which are used by the generated code. This separation of concerns greatly simplified our work, as it enabled all of the parallel execution and networking logic to be implemented separately from the high-level language features of XQuery.

In addition to being of interest to the workflow community, we believe that the ideas presented in this chapter may also be relevant to other XQuery implementors. In particular, support for web services is useful for a range of applications. In fact, there is such a natural fit between web services and the language facilities of XQuery that it seems surprising that this functionality is not already included in the language specification. We hope that our discussion of this idea will encourage the consideration of web service support in future versions of the language.

Chapter 6

Managing Parallelism

At first consideration, it may seem that with a side effect free language, effective parallel execution is simply a matter of detecting all expressions whose results are needed, and running them in parallel. Unfortunately however, the reality is not so simple. There were a number of performance problems we discovered during implementation that were not anticipated at the start of the project. In this chapter, we discuss these problems and how we solved them, as well as lessons learnt about the practicalities of using parallel graph reduction for distributed workflow execution.

Firstly, the choice between lazy and strict evaluation influences the degree to which opportunities for parallel evaluation can be detected automatically. Laziness delays evaluation as late as possible, while strictness allows evaluation to begin as soon as an expression is encountered, making it possible to trigger parallel evaluation more often. In the context of workflows, lazy evaluation also has the implication that arguments to a service begin evaluation only once the connection to the server has been established, resulting in connections being kept open much longer than necessary, and in certain cases leading to deadlock due to server-imposed limits on the number of simultaneous connections. Experiencing these conditions during our testing phase lead us to conclude that strict evaluation was the superior model to use for workflows.

Secondly, we discovered that naïvely initiating every service request as soon as its arguments became available would sometimes cause servers to become overloaded and start rejecting requests. To prevent this from happening, we added a throttling mechanism to the virtual machine which prevents it from making too many requests at once. A major challenge in implementing this was finding a way to dynamically determine the number of concurrent requests a server was capable of handling, in the absence of information about its available CPU and memory resources. We devised a solution to this problem which works by limiting the number of *outstanding* connection requests to a server, while permitting as many *accepted* connections as the server is able to support.

Thirdly, management of sparks must be implemented carefully to avoid introducing execution inefficiencies and poor load balancing. We discuss our choice of spark pool representation, which provides O(1) complexity for all important operations. Regarding load balancing of service requests, we identified conditions that can lead to machines being swamped with too much work, and developed a solution which enables service requests to be re-distributed to other idle machines where appropriate to balance the load. The development of this scheme was complicated by the need to distinguish between expressions corresponding to service requests and those corresponding to local computation, so that the latter could continue execution while service requests were being processed, in order to generate additional sparks required to keep other machines busy. We give experimental results showing the performance of our load balancing scheme.

Finally, we discuss the performance limits of our system in terms of the maximum frequency of service requests it can support. These limits are due to inherent properties of the TCP protocol, which requires each outgoing connection to a given server to use a distinct client-side port number, and prevents the reuse of port numbers for a period of between one and four minutes. This results in an upper limit of around 470 requests/second per server. It is only possible to go beyond this limit by forcing premature reuse of port numbers, which is technically a violation of the TCP specification. However, this limit is likely to be more than enough for normal usage.

In this chapter, we discuss all of these issues in detail. With the exception of Sections 6.1.1 – 6.1.4 and 6.3, the problems and solutions presented here are unique to workflows, and have therefore not been addressed by previous research into parallel graph reduction, which has only focused on programs that perform all of their computation internally rather than accessing external services.

6.1 Evaluation modes and detection of parallelism

As discussed in Section 2.4.1, an important choice in the design of a functional language is whether to use *strict* or *lazy* evaluation. With strict evaluation, function arguments are always evaluated prior to a function call occurring. With lazy evaluation, arguments are only evaluated if and when they are needed. Theoretically, all programs which produce a result under strict evaluation will also do so under lazy evaluation, but the converse is not true — programs written in lazy languages may depend on certain expressions not being evaluated, particularly those expressions that could lead to non-termination.

We chose to look at both modes of evaluation, to gain an understanding of how they influence the runtime behaviour of workflows, and to determine which is the most appropriate for our target usage scenarios. For this reason, NReduce permits the default evaluation mode to be specified as a configuration parameter, making it straightforward to compare the performance of a given program under both. The evaluation mode can also be specified on a per-expression basis for specific scenarios such as the creation of cyclic data structures, which relies on lazy evaluation, as discussed in Section 5.5.5.

The ability to specify a different evaluation mode for specific parts of a program is supported in some existing functional languages. Haskell, a lazy language, enables function applications to be marked as strict using the `$!` operator, and for constructor arguments to be annotated as requiring strict evaluation using the `!` annotation [168]. Scheme, a strict language, permits lazy evaluation to be used in specific cases via the `delay` and `force` operators [184]. In ELC, we permit function arguments to be specified as strict or lazy using the `!` or `@` prefixes respectively. All other arguments will default to the evaluation mode specified in the virtual machine's configuration.

Based on our comparison of the two approaches, which we shall discuss in the following sections, we concluded that strict evaluation was the only viable option for our needs. One reason for this is that lazy evaluation results in fewer opportunities for detecting parallelism, due to evaluation being postponed as late as possible. Another reason is that when invoking service operations, network connections are kept open for much longer than necessary, which in certain situations can lead to deadlock. Before discussing these problems, we will first explain the advantages of the two approaches, to illustrate why we considered both modes worthy of investigation.

6.1.1 Advantages of lazy evaluation

By delaying or avoiding evaluation, laziness has the potential to improve both the memory usage and execution time of a program. It is particularly useful in the context of data structures, where one part of the program lazily constructs a data structure, and another part traverses it. Only when the latter part of the program tries to access certain elements of the data structure will they be created. This enables some types of data structure production and consumption to occur in constant space, and for conceptually infinite data structures to be used, provided only a finite portion of them are ever accessed.

The following code illustrates a program fragment which, with lazy evaluation, only requires a constant amount of memory:

```
(foldr ++ "" (map callservice (range 1 n)))
```

The `range` function returns a lazily-constructed list of numbers, each of which is passed to a separate invocation of a service by `map`. The `foldr` function repeatedly applies the concatenation operator `++` to the contents of the list returned by `map`. Because the printing mechanism traverses the result of `foldr` sequentially, this causes `foldr` to traverse the list returned by `map` sequentially, which in turn causes the `callservice` invocations to occur sequentially. The effect of this is that only one item at a time is processed, requiring only a fixed amount of memory.

On the other hand, strict evaluation requires the complete list of numbers from 1 to n to first be constructed, after which all of the service calls are made, resulting in a list that is then passed to `foldr`, which computes the entire output before it is printed. Since each parameter must be fully evaluated before being passed to the relevant function call, this requires O(n) memory.

In the context of workflows, incremental stream processing can potentially be of benefit when transferring large volumes of data between services. With lazy evaluation, transmission of result data produced by a service can begin as soon as some of the data becomes available, even before the service has finished executing. Depending on how the receiver processes the data, it may also be possible to perform some of the computation on early parts of the data stream, prior to the full output of the producer arriving. This can improve execution times by beginning computation sooner, and reduce memory requirements by processing data in a streaming fashion instead of requiring it to be completely buffered in memory.

Finally, lazy evaluation permits the creation of cyclic data structures. For example, the following code creates a circular list containing the elements 1, 2, and 3:

```
letrec x = (cons 1 (cons 2 (cons 3 x)))
in     x
```

Since lazy evaluation does not require the parameters of `cons` to be evaluated at the time of the call, the tail of the list is set to the (as yet unevaluated) expression associated with the symbol `x`. Once this expression is eventually evaluated, `x` automatically refers to the expression's result. This would not be possible with strict evaluation, which would require `x` to be evaluated *prior* to the `cons` call, which is impossible, due to the circular dependency. Cyclic data structures are necessary for our XQuery implementation, as discussed in Section 5.5.5.

6.1.2 Advantages of strict evaluation

While lazy evaluation theoretically results in the least computation being performed, strict evaluation tends to be more efficient in practice, because it permits expressions to be directly evaluated. Every time an expression is delayed, the runtime system must allocate a new object to hold the expression and the variables it references; this allocation and subsequent garbage collection incurs a cost, which can be quite significant if this is done for every expression in a program. On the other hand, direct evaluation involves immediately executing whatever instructions are necessary to compute the result of a particular expression, without incurring this allocation cost. For this reason, efficient implementations of lazy languages generally make use of strict evaluation whenever possible, relying on strictness analysis (discussed in Section 2.4.1) to detect cases where it is safe to evaluate function arguments directly.

Automatic parallelisation is much more straightforward for strict evaluation, because any time two or more arguments are supplied to a function, they can all be treated as candidates for parallel evaluation. With lazy evaluation, this cannot be done safely, since it is not known at that point whether all the arguments will actually be used by the function. Strictness analysis can help in this regard by detecting cases where it is actually safe to evaluate arguments in parallel, but its necessarily conservative nature means that it cannot detect all cases.

The key difference between the two modes with regards to opportunities for parallelism is that while laziness delays evaluation as *late* as possible, strictness enables evaluation to begin as *early* as possible. This results in more entries in the spark pool on average, enabling more processors to be kept busy.

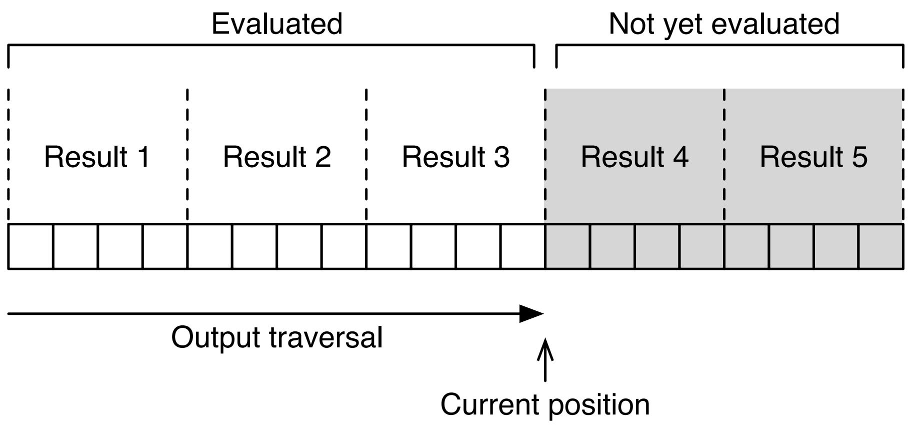


Figure 6.1: List traversal and evaluation during output

6.1.3 Detecting parallelism

Since we are aiming for automatic parallelisation, we want to be able to detect as many cases as possible where multiple function arguments can be evaluated in parallel. As mentioned above, this is straightforward for strict evaluation. For lazy evaluation, we must instead rely on strictness analysis. However, this does not detect all potential cases of parallelism. The implication of this is that in many cases, lazy evaluation affords fewer opportunities to evaluate expressions in parallel.

Consider the code from the first part of Section 6.1.1, which invokes a service call for each item in a list of numbers. Lazy evaluation executes this sequentially, because each portion of the result list is only demanded when it becomes time to write its result to output, which is only after the previous result has been written. Ideally, all of the service calls should be considered candidates for parallel evaluation, but this is not picked up by the strictness analyser, which only detects cases where the *start* of a list is needed.

Figure 6.1 shows the state of the list produced by `map` after the first three results have been printed. These values have been materialised in memory, because they were needed by `foldr` to produce the initial portions of the output stream. However, because it has not yet been necessary to obtain subsequent portions of the list, the remaining items do not exist yet in memory. The reference into the list from within `foldr` is a suspended expression which, when evaluated, will produce a cons cell whose head refers to the next service result, and whose tail refers to an expression that will produce the remainder of the list.

With strict evaluation, both arguments to cons would need to be evaluated before the cons cell can be constructed, and thus the whole list would need to be constructed before anything else can happen. This means that *all* of the service calls would be considered candidates for parallel evaluation almost immediately after execution begins, enabling the load balancing mechanism to distribute them across as many machines as are available.

6.1.4 Comparing parallelism in both cases

To illustrate the impact of the extent to which parallelism can be detected for the two modes of evaluation, we shall look at a program which contains two separate list processing phases:

```
(foldr ++ ""
       (map svc2
            (sort (\x.\y.- x y)
                  (map svc1 (range 1 n))))
```

This program starts by producing a list of numbers in the specified range, and then invokes a service operation on each, just like the previous example from Section 6.1.1. However, instead of printing these values out, it sorts the resulting list, and then invokes another service for each value in the sorted list. The results of these latter calls are then concatenated together into a string, which forms the output of the program.

Figure 6.2 shows the average processor utilisation of a set of four nodes when executing this program under both strict and lazy evaluation. In this case, an input size of 256 was used, with one second of computation for each service call, and all nodes were configured to process at most one service request at a time. Strict evaluation achieves nearly full utilisation the whole time, and completes in around 130 seconds. Lazy evaluation fully utilises all processors up to around 65 seconds, and then achieves an average of 25% utilisation (equivalent to one processor) for the remainder, taking around 310 seconds to complete.

What has happened here is that even with lazy evaluation, all of the values in the first list were demanded immediately, because sorting requires all input values to be evaluated before it can begin returning results. In the second phase of processing, the list is traversed sequentially, with only one item at a time being demanded. This causes lazy evaluation to achieve only sequential processing, for the reasons we have discussed.

The reason why both modes of evaluation perform identically up to the 65 second mark is that strict evaluation effectively occurs for the first stage in both cases, since all values in the list are needed for sorting. The strict evaluation mode has a short spike downwards at the 65 second mark, where it performs the sort, after which it makes the calls to the second service in parallel. It is exactly at this point in time that the lazy version also performs the sort, then drops down to a utilisation level equivalent to only one processor for the remainder of execution.

Figure 6.3 shows the performance differences as the number of processors increase. Both modes benefit from additional processors, because in either case the first portion of the program runs in parallel. However, the performance gains for the lazy version are much more limited, because of the sequentially executed portion of the program, which takes 256 seconds.

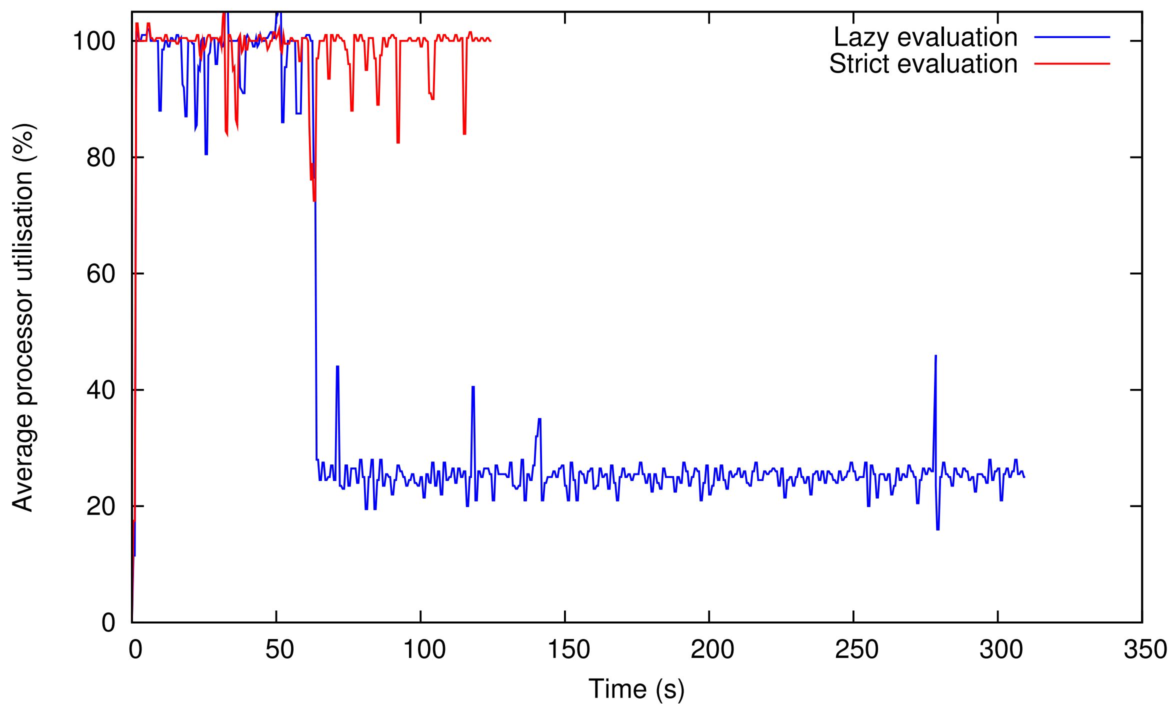


Figure 6.2: Average processor utilisation for strict and lazy evaluation (four processors)

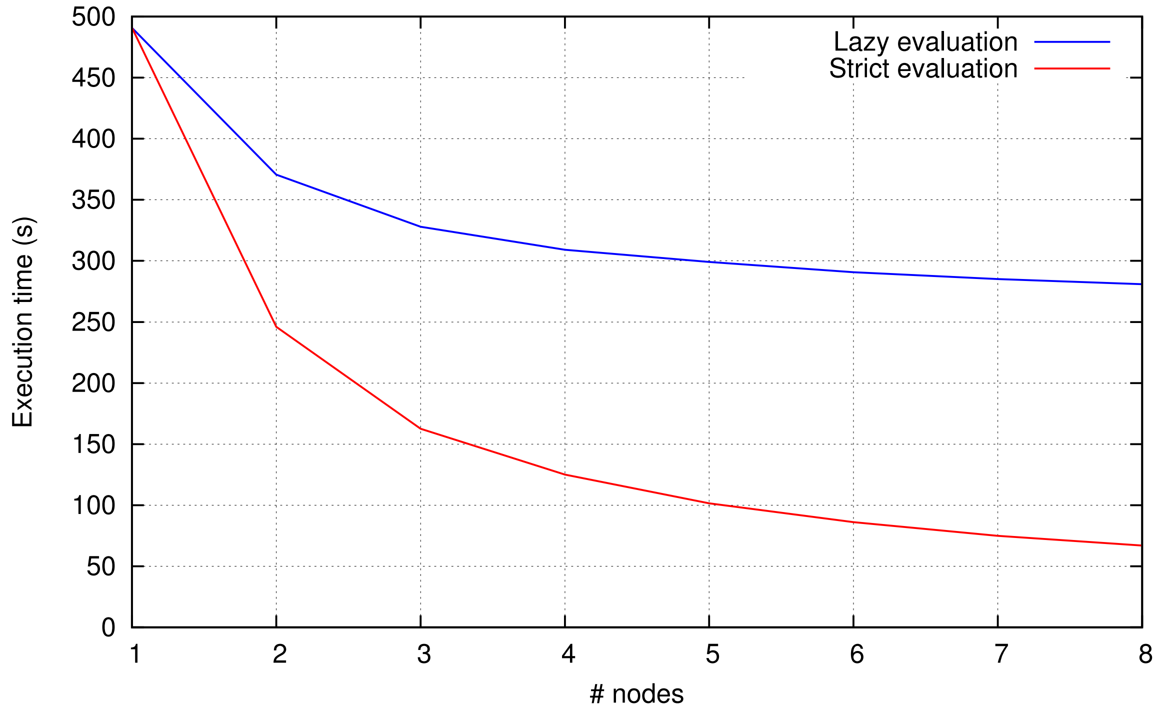


Figure 6.3: Performance improvements for strict and lazy evaluation

6.1.5 Laziness and service requests

An additional problem arises in the context of service requests, stemming from the fact that the contents of an input stream (sent from the client to the server) are usually only evaluated when the input stream is ready to be written. The `connect` function used to make a service request first opens a connection, then evaluates and writes the input stream, and then shuts down the connection for writing. This means that the connection is held open for the entire duration of the time that the input stream is being evaluated. If the contents of the input stream are derived from other service calls, then this can mean the connection will be held open while those other calls are being made.

Consider the following workflow, which makes calls to three services A, B, and C.

```
(A (B (C "test")))
```

Within the code, the tasks are realised as functions which call `connect`, defined in Section 3.5.8. Each task function supplies an input stream containing the necessary information to tell the server which operation to invoke, as well as the parameters that are passed in to the function. If we assume a simple text-based protocol, with the services all being provided by a machine called "server" on port 1234, then the functions would look something like the following:

```
A x = (connect "server" 1234 (++ "A " x))
B x = (connect "server" 1234 (++ "B " x))
C x = (connect "server" 1234 (++ "C " x))
```

When the workflow expression above is evaluated in outermost fashion, the following sequence of events occurs:

1. A opens a connection to the server and writes the initial part of the input stream, "A "
2. Evaluation of A's parameter is triggered, causing a call to B to start
3. B opens a connection to the server, and also writes an initial string
4. Evaluation of B's parameter begins, corresponding to the call to C
5. C opens a connection to the server, writes the complete input stream (which includes the string "test"), obtains a result from the server, then closes the connection
6. B finishes writing the result of C to its input stream, obtains a result from the server, and closes its connection
7. A finishes writing the result of B to its input stream, obtains a result from the server, and closes its connection

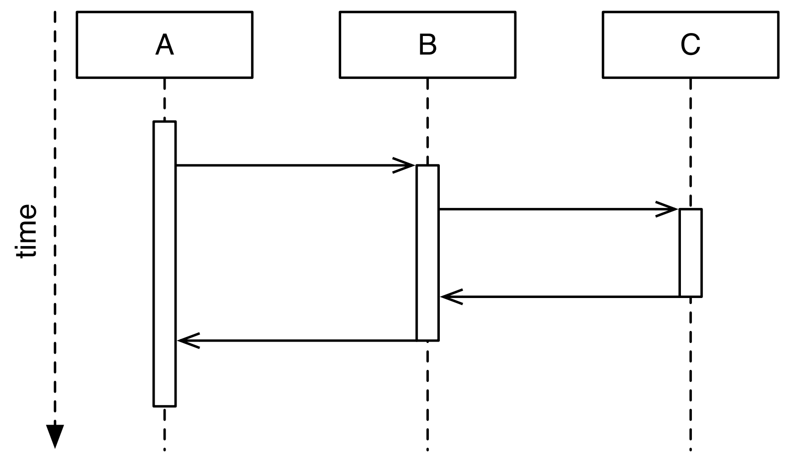


Figure 6.4: Connection durations for `(A (B (C "test")))`

An important point to note about this sequence is that the connection for A remains open for the whole time that B is being evaluated, and the connection for B remains open for the whole time C is being evaluated. This can be seen more clearly in Figure 6.4, which shows the events as a sequence diagram.

Intuitively, it does not seem necessary to keep these connections open for so long; it would be better to call C first, then B, and only then begin communication with the server for A's call. But the strictness analyser does not see this, so lazy evaluation of the expression causes the evaluation of each parameter to be delayed until after the caller's connection has already been opened. In the case of these three calls, this may seem fairly harmless, but when we increase the number of calls, the depth of the call tree, and the time of individual service calls, this can lead to more serious problems arising from a large number of connections being open.

Many servers have a limit on the number of concurrent connections they allow. If this limit is reached, and additional connections arrive, these new connections will wait in a queue until active connections have been completed. Consider what would happen if the server used by the three calls above had a limit of two active connections. The connections for A and B would be successfully accepted, and when the connection for C is attempted by the client, it would be placed in the queue.

Unfortunately, A and B cannot complete until the result of C has been obtained, and will keep their connections open until this occurs. But C cannot complete, because it is stuck waiting for its connection to be accepted, which will never happen, since A and B are holding their connections open. This is a deadlock situation, as depicted in Figure 6.5.

In order to avoid deadlocks like this, it is necessary to ensure that the parameters to a service call are evaluated *before* the connection is opened. This, in fact, corresponds to strict evaluation.

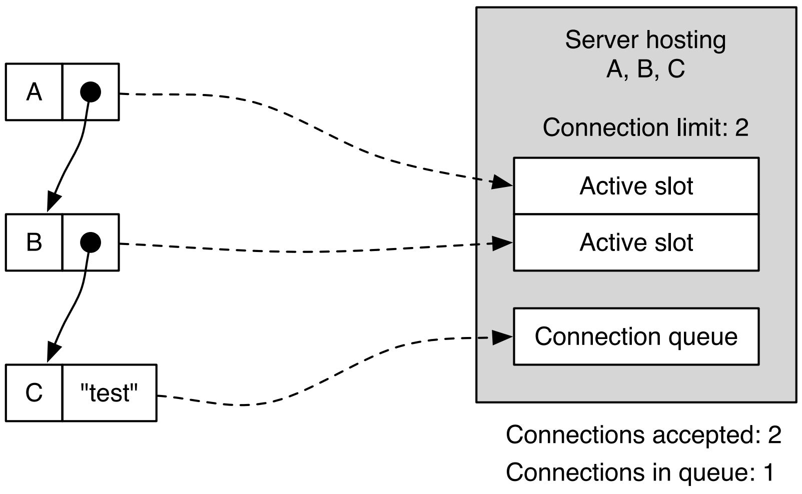


Figure 6.5: Connection deadlock: C is waiting for A and B to finish, which are waiting for C to finish

6.1.6 Summary

Our motivation for supporting lazy evaluation was that it could be used to support a streaming model of workflow execution, in which services incrementally process data as it arrives. However, experience showed that in the context of the type of workflows we are targeting, lazy evaluation causes more problems than it solves. The inherent inability of strictness analysis to detect all opportunities for parallelism means that in some cases, such as traversal of a list, services are invoked sequentially when they could instead have been invoked in parallel. Additionally, the way in which the number of open connections increases with the depth of the call tree can lead to deadlocks in cases where the total number of active connections allowed by the services being accessed is less than the depth of calls.

Based on our experiences as described above, we concluded that strict evaluation was the only practical option for workflows. However, due to the need to support cyclic data structures for the XQuery implementation, we incorporated into NReduce the ability to use programmer-supplied annotations to declare certain arguments as lazy, indicating that evaluation of the relevant actual parameters to these functions should be delayed. As explained in Section 3.5.6, formal parameters of a function can be marked as lazy by preceding them with an `@` symbol. This selective use of laziness in a small number of cases, with strict evaluation being the default, enables us to exploit parallelism effectively without sacrificing all of the benefits of laziness.

6.2 Limiting request concurrency

An important issue to consider when scheduling service calls is the number of concurrent requests that should be made at a time. This number is often difficult to determine, particularly when a load balancer is in use, because there may be any number of machines and processors available. Too few requests will result in the servers being under-utilised, while too many requests may cause the servers to become overloaded and start rejecting requests.

All machines have a practical limit on the number of requests they can handle at once, which depends on the amount of memory and processing power available. It is common for server administrators to set a concurrency limit as a configuration parameter, to prevent servers from running out of memory and potentially crashing if given too many requests. Any new connections established beyond this limit are queued until the accepted connections have been closed by the server. However, the size of this queue is typically very limited, and any new connection attempts that arrive when it is full are rejected [282].

Assuming appropriate server-side concurrency limits have been set, it is reasonable for a client to interpret the fact that a certain number of connections have been accepted on a given IP address as an indication that the server (or servers) at that address are capable of handling that number of concurrent requests. The client can also assume that connections that have been initiated but not yet accepted are either part-way through being established, or are waiting in the queue. By limiting the number of these *outstanding* connections, but not the number of *accepted* connections, a client can maximise utilisation of servers without requesting more work than the servers can handle.

This technique is analogous to the flow control [281] that occurs within the context of an established TCP connection. A sender uses the *window size*, advertised by the receiver, to limit the amount of unacknowledged data that may be outstanding. This enables a fast sender to avoid overloading a slow receiver. However, TCP does not provide any built-in mechanism to limit the rate at which connections are established between a client and server, which is why we have had to come up with our own way of doing this which operates at the application level.

6.2.1 Available capacity

In general, there is no way for a client to know in advance how many concurrent requests a server available at a particular IP address may be capable of handling. This depends on what resources are available at that address, which may range from a single machine with only one processor, to a large server farm with hundreds of multiprocessor servers placed behind a front-end load balancer. It also depends on the amount of computation time and memory used by a service. The number of concurrent connections can also be limited by a configuration parameter, which the client has no way of knowing.

Consider a situation in which a client is accessing a single server, as shown in Figure 6.6. The server is configured with a maximum of six accepted connections, and a queue size of four. The maximum number of accepted connections are in use, and one new connection resides the queue. In this case, the client may initiate up to three additional connection attempts before the server will start refusing further requests. If the limit is exceeded, the

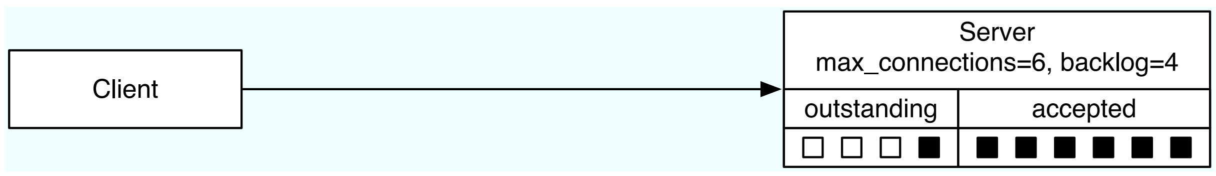


Figure 6.6: Client interacting with a single server

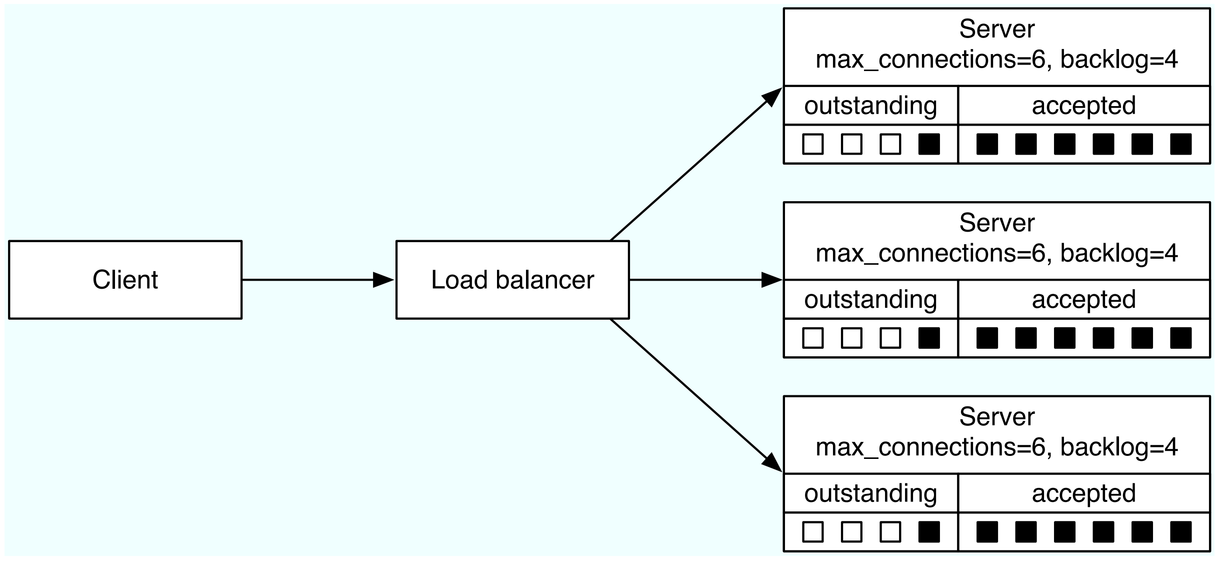


Figure 6.7: Client interacting with multiple servers located behind a load balancer

requests will be ignored, causing the additional connection attempts to eventually time out.

Figure 6.7 shows another scenario, in which the IP address the client connects to is a load balancer with three servers behind it. Each server is configured identically to the previous case, making for a total of up to 18 accepted connections, with 12 outstanding. In this case, the client may quite reasonably have 18 concurrent requests in progress, because the servers are capable of handling this many. However, because the client does not know how many servers are actually available, it should still avoid having too many outstanding requests. For this reason, it should err on the side of caution, and limit itself to having only a small number of outstanding connections at a given point in time, to cater for the possibility that there may be only one machine at a given IP address.

For the experiments described in Chapter 7, we used a limit of three outstanding connections at a time. All machines were dual-core, so this permitted both cores to be kept busy with a request, with one extra request available to fill the time between another ending and a replacement arriving. This number is also low enough to avoid filling the backlog queue on the servers.

One potential downside to limiting the number of outstanding connection attempts is that when a large number of short service calls are being made, this limit may result in suboptimal utilisation of the servers. In Figure 6.7, there are actually 12 queue slots available for new connections, and if the load balancer ensures even distribution of requests, then all of these slots could be in use at a given time. Short per-request processing times require larger numbers of incoming connections to be in the pipeline to enable full utilisation of machines. Limiting the number of outstanding requests to a small number can prevent this from being achieved. Allowing this limit to be set as a configuration option on the client may be desirable in situations where the user knows that a large number of servers are available, though in our implementation we stick to a fixed limit of three. Techniques to automatically determine this limit seem difficult to achieve in the absence of out-of-band mechanisms to determine queue sizes, since TCP provides no inherent capability for clients to determine the queue size of server. One possible approach may be to dynamically adjust the limit based on a function of server response (acceptance) time and service completion time. We leave this for future work.

6.2.2 TCP connection establishment

In order for a client to limit the number of outstanding connection attempts (those which have been initiated but not yet accepted), it is necessary for it to know when a connection is accepted. Unfortunately however, TCP only notifies the client when a connection is *established*, not when it is *accepted*. There may be an arbitrarily long delay between these two events, while the server process[1] finishes handling a previously accepted connection. The client therefore cannot assume that an established connection is actually being handled by the server process, and must postpone additional connection attempts. This implies that an alternative strategy must be used to determine when acceptance is known to have occurred. Before discussing our solution to this problem, let us first examine the process of connection establishment.

TCP connections are established using a so-called *three-way handshake* [282]. This is a sequence of three segment exchanges between the client and server that together comprise their first phase of interaction, before any application data is transmitted. This logic is implemented in the kernel's TCP stack, with the server process only receiving notification after the handshake is completed.

Before connections can be established, the server process must create a socket and perform a *passive open* on it. This is achieved using the `listen` system call, which is called by the server process when it is ready to receive requests on a particular port. Once it has done this, clients may start establishing connections, by performing *active opens* using the `connect` system call.

The first step in the handshake occurs when a client sends a SYN segment to the listening socket. This segment indicates to the server that a client wishes to establish a connection, and supplies the client's IP address, port, and initial sequence number. The server then responds with a SYN/ACK segment, acknowledging the connection request and sending its own initial sequence number. Finally, the client sends an ACK segment to the server

[1]There is a distinction between the terms *server* and *server process*. The server is the machine on which the server process (application) is running.

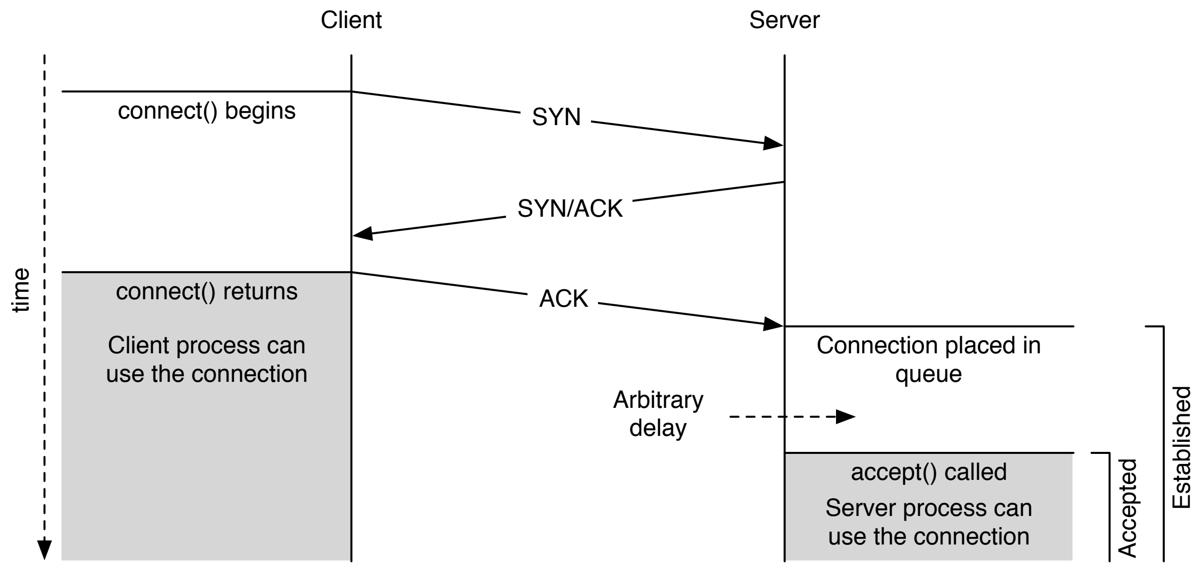


Figure 6.8: Three-way handshake

acknowledging its receipt of the SYN/ACK. From this point onwards, the connection is considered established. This process is shown in Figure 6.8.

On the client side, the `connect` system call will return as soon as it has sent the ACK segment. As far as the client is concerned, the connection appears as if it is all set up and ready to be used, and the client may start sending data. The TCP stack within the server's kernel may receive this data, and buffer it up to a certain limit. Only when the server process invokes the `accept` system call will it gain access to the connection and the initial data, and be able to send back response data. The call to `accept` is purely an internal detail involving the server and server process, and does not involve any notification or interaction with the client. It is for this reason that the client has no way of knowing exactly when it has occurred.

6.2.3 Determining when a connection has been accepted

On the client side, the closest we can get to knowing that a connection has definitely been accepted is when we have started receiving data on it. For the server process to have sent this data, it must have first accepted the connection. Between the time at which the `connect` system call returns and the first byte of data is received, the client cannot be certain whether the connection is in the queue, or has been accepted by the server process but is yet to have any data sent back. Our implementation thus relies on at least one byte of data having been received from the server process before considering the connection to have been accepted.

Some protocols, such as HTTP, do not require the server process to send any data initially after accepting a connection. A client sends a request, this request is then processed, and then a response is sent back. If the request processing takes a long time, then the connection will remain in an unknown state for a significant amount of time. During this period, the client cannot assume that the connection has been accepted, and must therefore hold off on making any further connection attempts beyond the (small) maximum limit discussed in Section 6.2.1, to avoid overloading the server. However, this can severely limit the amount of parallelism that can be exploited in cases where the server has a large number of processors.

Our solution of waiting for data to be received by the client before it considers the connection to be accepted will only work for protocols in which the server process sends back some initial data prior to the client requesting that work be performed by the server. Some protocols, such as IMAP [77] and POP [236], initially send a welcome message before reading input from the client. The client can use the receipt of this message as acknowledgment that the connection has been accepted for processing.

Although HTTP does not utilise welcome messages, we can get a similar effect by using persistent connections to send two requests over each connection. The first is a simple HEAD request which causes the server to reply quickly without performing any compute-intensive work, enabling the client to detect acceptance as soon as possible. The second is the actual request for the service operation, which can involve an arbitrary amount of input data and server-side computation. Both requests are sent immediately after connection establishment; the client does not wait for the first response to come back before beginning transmission of the second.

Regardless of how long the second response takes to arrive, the client knows that the connection has already been accepted as soon as the first response comes back. The client can then safely initiate another connection to the same IP address, with the goal of exploiting additional capacity on the server. If the server capacity is greater than the client's demands, the frequency with which connections are initiated is a function of the latency between the two machines, and the time taken by the server process to respond to a HEAD request. If client demand outstrips server capacity, then the frequency of new connections will also be affected by the time taken to complete the requested service operations.

The web service invocation mechanism used by our XQuery implementation, described in Section 5.3, uses this technique.

6.2.4 Client-side connection management

We have established the need to keep track of which connections are outstanding and which have been accepted, and to make this distinction based on whether or not any data has been received from the server for a given connection. We have also discussed the need to limit the number of connections to a given server that may be outstanding at a given point in time, to avoid filling the backlog queue.

To implement this logic, we model connections using a finite state machine, depicted in Figure 6.9. All connections between a client and a server are managed according to this state machine, and certain transitions are used to keep track of when a client enters or

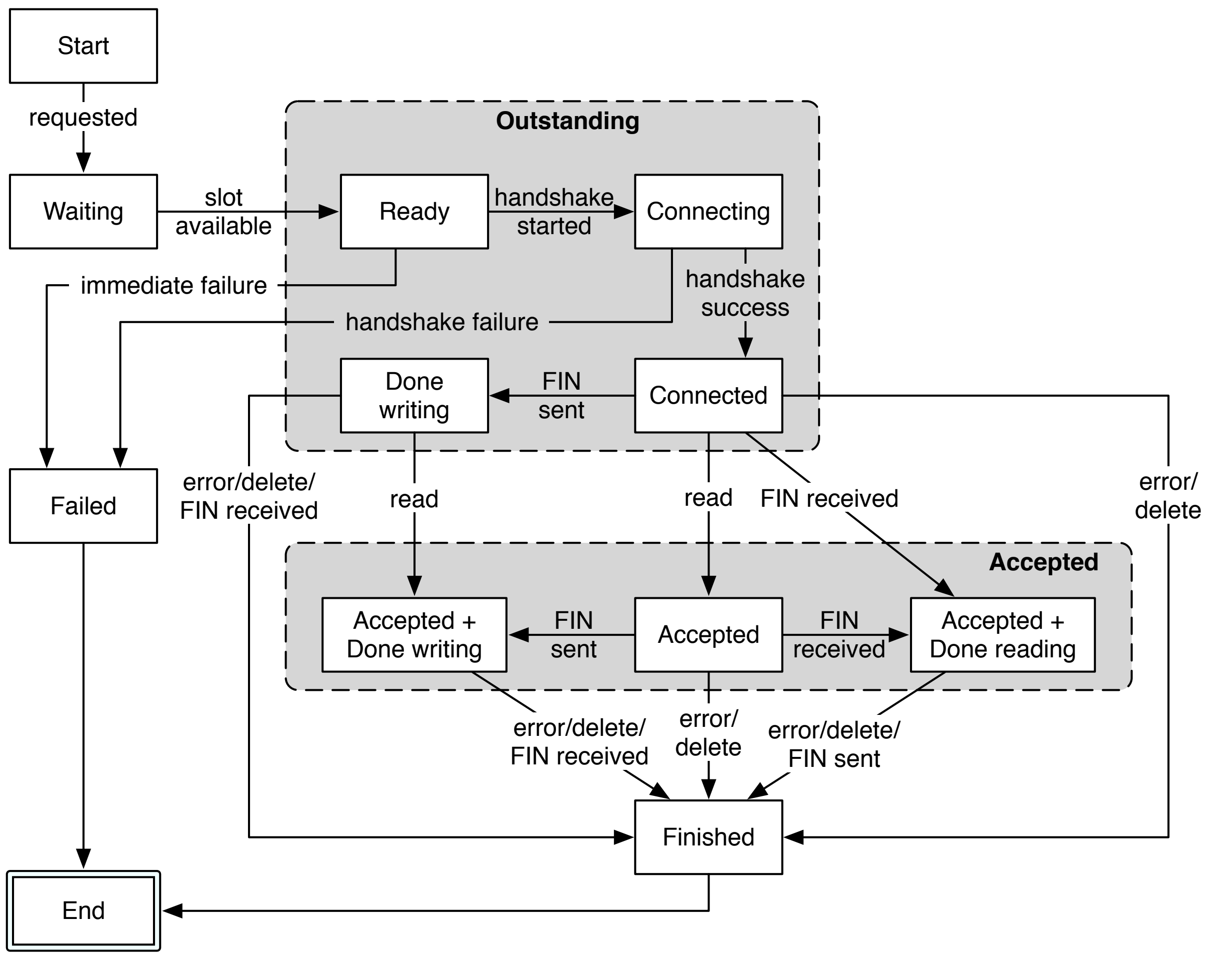


Figure 6.9: Application-level state machine for connection objects

exits the *outstanding* or *accepted* phases, shown as shaded boxes in the diagram. Although this has similarities with the protocol-level state machine defined for TCP [252], this state machine refers to application-level connection objects which are maintained by the I/O thread of the virtual machine.

Initially when a workflow tries to connect to a service, a new connection object is created, and immediately transitions to the *waiting* state. This state indicates that the connection has not yet begun the three-way handshake, and the runtime system will wait until the number of outstanding connections to that particular server falls below a threshold (which in our implementation is set at three) before initiating the connection. As soon as this is the case — possibly immediately — the connection transitions to the *ready* state and an invocation of the `connect` system call will be made asynchronously. If this fails immediately, the connection transitions to *failed*, otherwise it goes into the *connecting* state. A connection failure causes a notification message to be sent back to the execution thread, which aborts execution of the program; our current implementation does not support fault tolerance.

In the case of a successful connection, the `connect` system call indicates completion as soon as the three-way handshake has finished, which may be some time before the server process actually invokes the `accept` system call for the connection. Beginning from the time at which `connect` returns, assuming the connection was successful, the client may start using the connection object by writing data to it and attempting to read. Some amount of data may be written prior to the server process's acceptance, in which case the data will be buffered by the TCP stack on the other end and made available to the server process once the connection is eventually accepted.

As soon as the client reads at least one byte of data from the connection, it knows that it has been accepted, and therefore no longer counts towards the total number of outstanding connections to that server. As soon as this occurs, the I/O thread will check if there are any other connection objects for the same server currently in the *waiting* state; if so, it will pick the first one and cause it to transition to the *ready* state, causing it to begin the three-way handshake.

Our implementation supports TCP's notion of *half-closed* connections, in which one side has indicated that it has finished sending data, but the other side is still sending data back. This is achieved by having the sender invoke the `shutdown` system call, which causes a FIN segment to be sent to the recipient, which will subsequently receive a return value of zero from the `read` system call once it has read all of the buffered data. The state machine explicitly models both the cases of the client having finished its transmission of data to the server, and having received the last portion of the data stream from the server. The connection is only closed once both sides have finished sending data.

The use of a finite state machine to model the representation of connections within the I/O thread enables the notion of outstanding and accepted connections to be modeled as groups of states. For each server, a count is maintained of how many connections are in each of these groups. The state transition logic updates these counts whenever a connection object enters or exits either of these groups of states, and ensures that the limit on the number of outstanding connections is enforced. No limit is placed on the number of accepted connections.

The scheme discussed here is effective at exploiting the maximum parallelism that a server or group of servers at a particular IP address are capable of providing, without overloading them with too many requests. This mechanism is implemented within the I/O thread of the virtual machine, which, as discussed in Section 4.5.2, is distinct from the execution thread responsible for running the actual program.

Within the execution thread, parallel execution of service requests and other expressions is controlled by the *spark pool*, which contains frames that are considered candidates for parallel evaluation. In the following section, we discuss efficiency concerns relating to management of sparks.

6.3 Spark pool management

A key aspect in the performance of a parallel graph reducer is the costs associated with sparking expressions. A *spark* is a frame whose result is known to be needed by the program, and is thus eligible for evaluation in parallel with other such frames. The sparking and frame management logic of our virtual machine were described in Sections 4.3 and 4.4. Sufficient sparking must be performed in order to exploit parallelism, and thus it is important to minimise the degree of overhead this contributes to program execution.

In languages that parallelise only internal computation, the performance of code executed by the language itself is the main factor that determines total execution time. Because of the costs associated with sparking, these language implementations minimise the regularity with which sparking occurs by only doing so where explicitly directed to by the programmer. However since we are targeting workflows, in which most computation is performed *externally*, we can trade off some degree of local execution speed for the benefits of automatic sparking, making the programmer's job easier. As long as the majority of computation involved with the workflow is carried out by coarse-grained service operations, the overheads of additional sparking will not significantly impact overall execution time.

6.3.1 Costs of sparking

There are two main ways in which sparking affects performance, which we categorise as *direct* and *indirect* costs. Direct costs are those incurred by the operations required to manipulate sparks, such as adding and removing expressions from the spark pool, and locating sparks when a processor becomes idle. These can be optimised by choosing appropriate algorithms and data structures to minimise the amount of time each of these operations takes, as we discuss in the next section. Indirect costs are the missed opportunity costs of compile-time optimisations that could have been performed were it not for the need to spark expressions.

Optimised implementations of graph reduction based on sequential abstract machines achieve much of their efficiency by avoiding allocation of temporary graph cells whenever it is known that it is safe to evaluate an expression directly. Normally, it is only necessary to explicitly construct a graph for the expression when it is subject to lazy evaluation, since the expression cannot be evaluated immediately. Sparking imposes the same form of overhead as lazy evaluation, because adding an expression to the spark pool implies that it must be explicitly allocated as a graph node. This allocation, and subsequent garbage collection, adds to the program's execution time.

Sparking, as well as lazy evaluation, also carries a secondary cost: when it eventually becomes time to use the result of the expression, the program must check to ensure that the expression has been evaluated before continuing. In our implementation, this is done using the EVAL opcode. This opcode checks if a referenced graph node corresponds to an unevaluated expression (i.e. a frame) and if so, removes it from the spark pool if necessary, before forcing evaluation of it. These checks would not be required if it was already known for certain that the expression had been evaluated.

6.3.2 Minimising direct costs

There is a limit to what we can do about the indirect costs, assuming that we want as many expressions as possible to be sparked. However, we can influence the impact of the direct costs by taking care to ensure that spark-related operations are implemented efficiently. These direct costs are primarily affected by the choice of representation for the spark pool. The desired properties that a representation should meet are as follows:

1. O(1) insertion (when expressions are sparked)
2. O(1) removal (when sparks are removed, before being evaluated)
3. O(1) search time when locating a spark to activate (when the processor becomes idle)
4. Avoid allocating separate objects for each entry in the spark pool (on top of the frame allocation that is already required to represent a spark)

There are several possible data structures that could be used to represent the spark pool:

- **A field in each frame object indicating whether or not the frame is sparked.** This avoids explicit representation of the spark pool, instead relying on a traversal of the heap to be performed whenever sparks are searched for. Insertion and removal are both very cheap, since they only involve modifying a single field. Separate allocation is also avoided, since the sparked/not sparked status is stored directly within each frame.

  This technique is what we originally used in NReduce. It was based on the assumption that searching for sparks would be a relatively rare occurrence, and thus it was acceptable to pay a slightly higher cost when searching for sparks, in order to gain the benefit of making sparking and unsparking very cheap. However, scalability testing showed that for workflows involving large numbers of service calls, the large heap sizes and frequent search operations limited performance.

- **A circular buffer.** This gives O(1) insertion and search time, since when searching for a spark, we simply need to look at the beginning of the buffer. Separate per-spark allocation is also avoided, since only a single buffer needs to be allocated, which is used to store pointers to all of the sparks. However, removal of a spark that resides part-way through the buffer incurs an O(n) cost.

  This approach is used by GUM [300], which relies on programmer-supplied annotations to perform sparking, which generally results in fewer sparks than for automatic parallelisation. GUM only removes sparks from the pool when they are activated due to an idle processor — normal evaluation of a previously sparked expression leaves the reference in the pool. When an idle processor searches for a spark to activate, it must check and discard sparks that have since been evaluated.

Since our implementation generates large numbers of sparks, this solution would lead to similar problems as the previous technique, since each search would have to inspect many entries. We therefore wish to remove frames from the spark pool whenever they begin evaluation; the O(n) cost of a circular buffer in this case is prohibitive.

- **A singly-linked list.** This is the technique used for the runnable list, for which we only ever need to add and remove items from the start, both of which are O(1) operations. Searching for the first available spark is cheap, because it simply involves looking at the head of the list. However, removing an item from part way through a singly-linked list is O(n), which would lead to the same unacceptable performance overhead as with a circular buffer.

- **A doubly-linked list.** This allows both insertion and removal from anywhere in the list with O(1) complexity, given a reference to a spark that is already in the list. It is this technique that we therefore chose to use in NReduce. GranSim [207] also uses this technique for spark pool management. As with singly-linked lists, separate allocation can be avoided by storing pointers to adjacent entries directly within the frames themselves.

  Another advantage of using a doubly-linked list is that we can add sparks to either end, depending on the circumstances in which we want them to be evaluated. As we discuss in Section 6.4, we use this ability to support the notion of *postponed* sparks, which are only activated when the machine is not fully utilised. This is important for load balancing when the virtual machine runs in a distributed manner across multiple nodes.

  We also maintain a sentinel node, which is always present at the list and permanently acts as both the first and last item. The insertion and removal operations can thus avoid having to deal with the beginning and end positions as special cases, since these operations are never performed on the sentinel node.

The spark pool is used as a source of frames eligible for parallel execution, which is consulted when the runnable list becomes empty, or a work request arrives from another machine. When such a request is processed, multiple sparks are sent to the requester, in case a single spark does not contain sufficient work to keep the machine busy for a significant amount of time. In section 6.4.7, we discuss how we determine the number of sparks to send in response to a work request. First however, we examine what is necessary to achieve a good distribution of sparks between machines when the virtual machine is running in its distributed mode.

6.4 Load balancing of requests during choreography

When using choreography to execute a workflow, it is the virtual machine nodes, rather than an external load balancer, which are responsible for deciding which service requests will be sent to which machines. Instead of simply establishing all connections to an external load balancer, choreography makes use of the work distribution mechanisms described in Section 4.7.4, which assign sparked frames to idle machines in response to work requests. Some of these frames may cause service invocation, which for choreography occurs on the same physical machine as the virtual machine node running the frame.

6.4.1 Processing assigned sparks

When an idle machine receives a response to a work request it previously sent out, there are two options for handling the frames contained in the response message:

1. Run only the first frame, and place all others in the local spark pool
2. Run all of the frames

At first glance, the first option seems to be the most logical, because a machine only needs one runnable frame in order to keep busy. If this frame later completes or becomes blocked, another entry from the local spark pool can be activated without having to send out another work request. If the frame instead causes a large amount of local computation to be performed, then the remaining entries in the spark pool can be used to serve work requests that subsequently arrive from other idle machines. Redistributing sparks in this manner can help to reduce load imbalance by enabling machines that are assigned too much work to offload some of their sparks to others.

Our execution model does not support the migration of arbitrary runnable or blocked frames. This means that once a frame has started running on a particular machine, it must stay there. The second option could thus potentially lead to load imbalance, if it turned out that one machine was initially assigned too much work, and was unable to redistribute it to other machines that later became idle after finishing their initial batch of work.

If we were only scheduling internal computation, then this analysis suggests that option 1 would be the best. However, if all of the sparks correspond to service calls, then each time a frame is run, it will begin a connection attempt and block. This will cause the next spark to be moved from the spark pool to the runnable list, which will do the same thing. This process will continue until all of the sparks have been activated, and are all in the process of making asynchronous connection attempts. The first few connections will be established almost immediately, but the frames attempting to establish the other connections will remain blocked, unable to migrate. This means that for service calls, both of the above options have the same effect, which is to cause a load imbalance between machines. Figure 6.10 shows an example of how this can occur.

6.4.2 Determining when a machine is idle

In addressing load imbalances caused by the situation described above, we need to look at what it means for a machine to be considered idle, i.e. a candidate for having additional

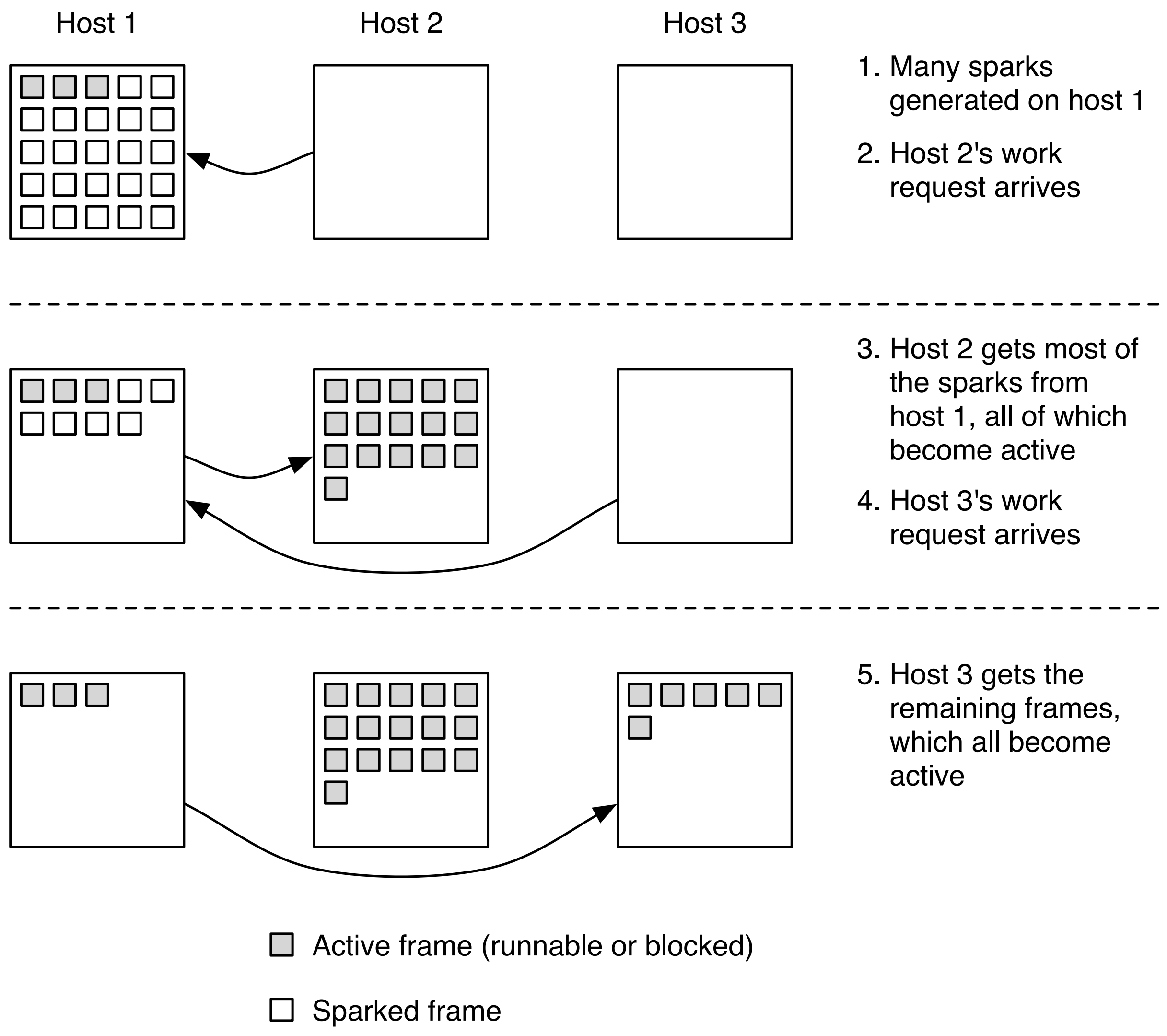

Host 1
Host 2
Host 3
1. Many sparks generated on host 1
2. Host 2's work request arrives
3. Host 2 gets most of the sparks from host 1, all of which become active
4. Host 3's work request arrives
5. Host 3 gets the remaining frames, which all become active
Active frame (runnable or blocked)
Sparked frame

Figure 6.10: Load imbalance caused by spark distribution

sparks activated. In traditional implementations of parallel graph reduction, which only deal with internal computation, it is sufficient to consider a machine idle if it does not have any runnable frames. However, in the case of service choreography, it is possible for there to be no runnable frames, but one or more local service calls in progress. A machine can thus be busy executing these, in which case it would seem sensible to avoid giving it extra work.

For multiprocessor machines, achieving full utilisation requires multiple service calls to be in progress at the same time, assuming each is handled by a single thread. This should be taken into account when deciding if a machine should be given more work; if there are fewer service requests in progress than the machine is capable of handling, then more can be made. We adopt the approach of allowing one more concurrent request than there are physical processors in the machine — this gives additional load that can be scheduled by the operating system during the delay between the completion of one service request and the beginning of the next, which may involve another work request being sent out. Since the hardware setup we are using consists of dual-processor machines, we use a maximum of three concurrent requests for all of our tests.

The condition a virtual machine node uses to determine if it is idle is the following:

$$(nrunnable = 0) \text{ and } (nrequests < max)$$

The next issue is what do to when a machine is not idle. Intuitively, it would seem that any additional frames residing in the local spark pool should not be activated until the number of requests drops below the maximum. However, this introduces another problem: it prevents opportunities for generating additional parallelism required to keep other machines busy.

Frames are added to the spark pool by the SPARK instruction, which is generated by the compiler whenever it can determine that an expression will be needed at some point in the future. If internal computation is postponed until the number of service requests drops below the maximum, this can prevent the generation of additional sparks which could otherwise have been distributed to other idle machines. In some circumstances, this can mean that the rate at which additional work is generated is the same as the rate at which local service requests complete, meaning that there are only enough sparks to keep a single machine busy.

6.4.3 Deciding which sparks to run

To achieve sufficient levels of parallelism, we want to activate frames from the spark pool that could generate additional work, while avoiding initiation of more concurrent service requests than the machine is able to handle. This implies that it is necessary to distinguish between the two types of sparks, and have a mechanism which permits the virtual machine to choose which types of sparks can be activated based on whether or not it is currently idle. An idle machine should be able to activate either sparks representing internal computation or those corresponding to service requests. A busy machine should only activate sparks that perform internal computation.

A complication with this is that in general, it is not possible to determine whether a sparked frame will invoke a service request or not just by inspecting the frame. If the code address associated with the frame is equal to the start of the built-in `connect` function, and the first parameter is `localhost`, then this definitely indicates a local request. However, the function may instead be a call to another function, which in turn calls `connect`. In the latter case, the frame may perform some arbitrary computation before deciding whether or not to call the service, meaning that simple static approaches to inspecting the frame graph cannot determine whether it will call `connect` or not. The only way to reliably determine whether a frame will invoke a service call or not is to run it. But if we do this, we need to be able to keep open the possibility of redistributing the frame to another idle machine if it turns out that the frame does involve a service request, and that request cannot be immediately satisfied.

6.4.4 Postponing connections

We mentioned earlier that in general, a frame that has started running can never migrate. However, in order to deal with this particular case, we make a special exception for calls to `connect`. Whenever an attempt is made to establish a new connection to `localhost`, the built-in `connect` function checks how many local service requests are already in progress. If this is already at the maximum, it *postpones* the connection attempt by removing the current frame from the runnable list, and placing it back in the spark pool. This makes it possible for the frame to be redistributed to an idle machine if a work request is received before the frame gets a chance to run locally. Figure 6.11 depicts a scenario in which this occurs.

The strategy used to allow local computation to continue while postponing additional service requests is to have an idle machine run *all* frames it receives in response to a work request — which is the second option discussed in Section 6.4.1. Any frame which does not cause the built-in `connect` function to be invoked will be allowed to continue, while those which try to connect to the local machine will be attempted, and placed back in the spark pool if necessary. These frames will either be tried again when the current batch of service requests completes, or be redistributed to other machines.

Placing service request frames back in the spark pool presents a problem: it is necessary to ensure that they are not attempted again until there is a possibility they will succeed. Without this, the frames would continually be attempted over and over again, wasting CPU cycles and preventing the virtual machine from spending time on legitimate work. To avoid this situation, the spark pool is split into two parts: one containing *regular* sparks, which have never been run, and the other containing *postponed* sparks, which are service requests that have been attempted and delayed.

6.4.5 Spark addition and activation

By having the spark pool arranged in two sections, the virtual machine can selectively activate sparks depending on the state that it is in. If it is idle, meaning that it has no runnable frames and the number of local service requests is less than the maximum, then either regular or postponed sparks can be activated. If, on the other hand, the service is already busy with the maximum number of requests, then only regular sparks can be activated. Although a regular spark may actually cause a connection attempt to be made, this will only happen once, and if it cannot immediately succeed then it will be placed in the postponed section of the spark pool.

As discussed in Section 6.3.2, the spark pool is stored as a doubly-linked list. This makes it easy to both add and remove frames from either end of the list, a fact which aids usage of the two different sections. Regular sparks always appear in the first portion of the list, and postponed sparks in the second. Figure 6.12 shows an example of this arrangement, with six regular sparks and four postponed ones. When a response to a work request is received, or the SPARK instruction is executed, the sparks are marked as regular and are

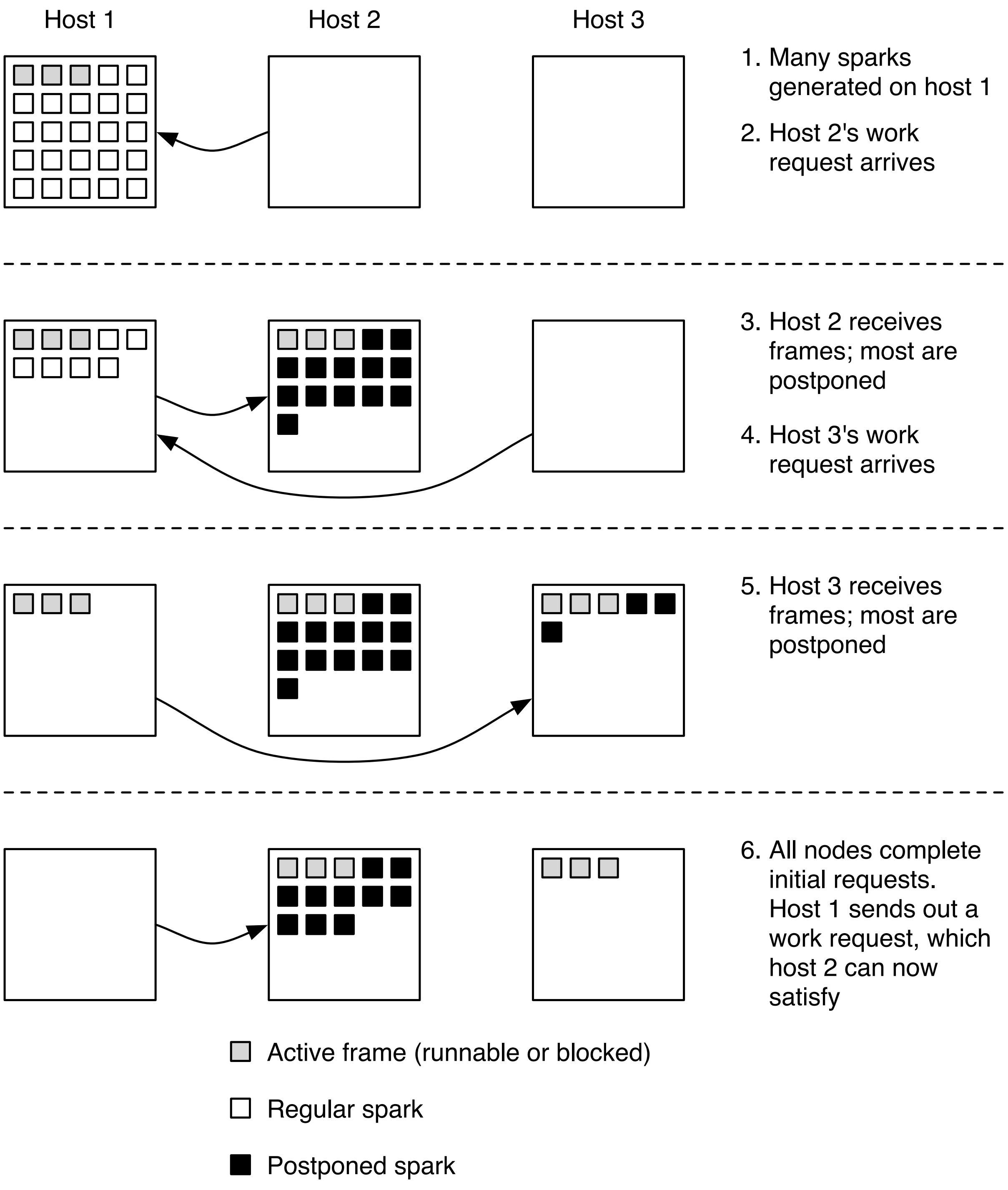

Host 1
Host 2
Host 3
1. Many sparks generated on host 1
2. Host 2's work request arrives
3. Host 2 receives frames; most are postponed
4. Host 3's work request arrives
5. Host 3 receives frames; most are postponed
6. All nodes complete initial requests. Host 1 sends out a work request, which host 2 can now satisfy
Active frame (runnable or blocked)
Regular spark
Postponed spark

Figure 6.11: Redistribution of postponed sparks

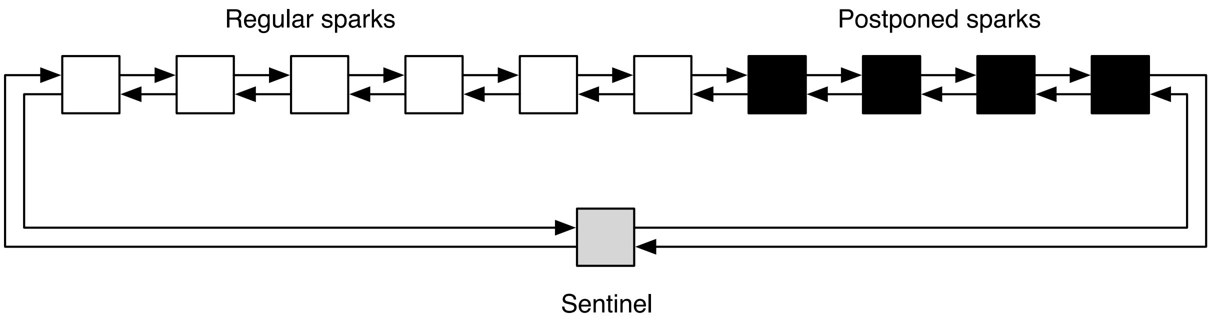


Figure 6.12: Spark pool

added to the beginning of the list. When a frame is placed back in the spark pool by `connect`, it is marked as postponed and added to the end of the list.

Whenever a virtual machine node wishes to activate a spark from its local pool, it inspects the front of the list. If the machine is idle, it activates the first entry in the list, regardless of whether it is regular or postponed. Otherwise, it will only activate the entry if it is a regular spark. In both cases this involves only O(1) complexity, because if the first entry in the list is marked as postponed, then this means that all of the entries must be postponed, and no further searching is required.

When a work request is received from an idle machine, the sparks to be sent back are taken from the end of the list, causing postponed sparks to be given first preference. These represent work that the local machine is definitely not yet ready to perform, so it is worthwhile giving them to another machine. For regular sparks, there is a chance that these could be executed locally in the near future, so these are only distributed if no postponed sparks remain.

6.4.6 Example

The following code excerpt uses data parallelism to invoke a service call for each item in a list of 4,096 elements. The service is invoked by sending it a request in the form "*c n*", where *c* is the number of milliseconds worth of computation to perform (128 in this case), and *n* is a numeric value taken from the list, which is simply echoed back by the server. This serves as a simple demonstration of processing items in a list using parallel service calls.

```
callservice n = (connect "localhost" 5000 (++ "128 " (ntos n)))

results = (map callservice (range 1 4096))
```

The `map` function is defined as follows:

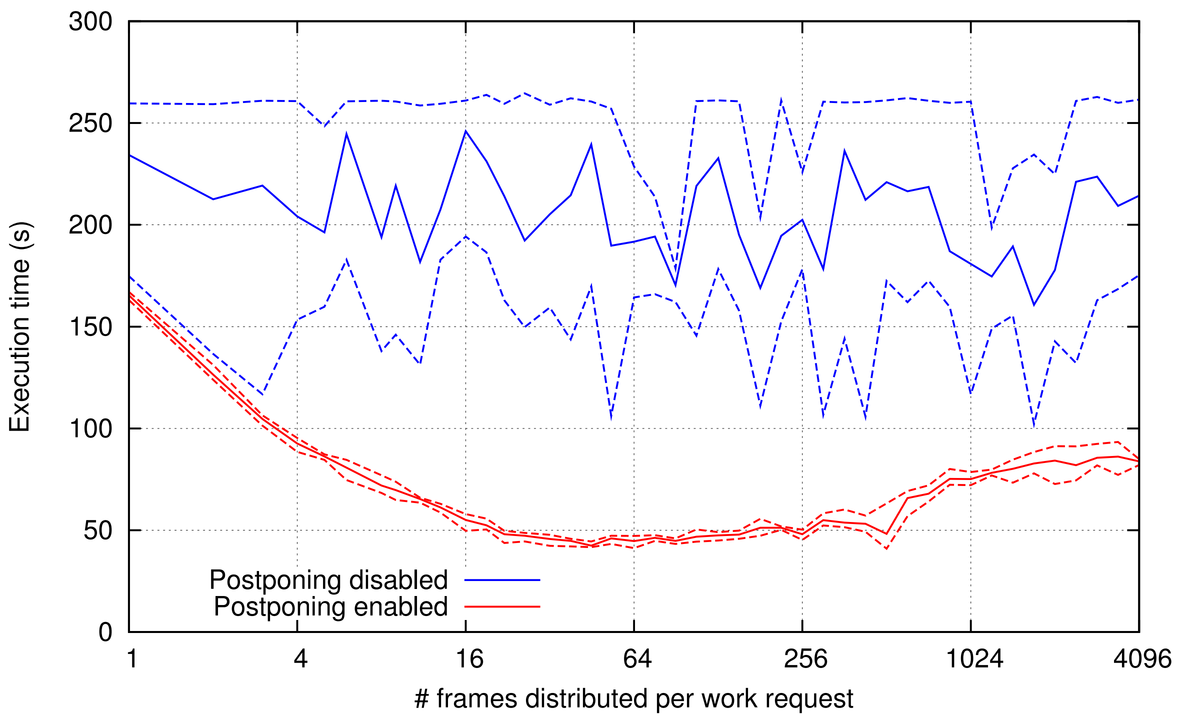


Figure 6.13: Performance with postponing enabled and disabled. Dashed lines represent the minimum and maximum, and solid lines represent the average.

```
map f lst = (if lst
                (cons (f (head lst))
                      (map f (tail lst)))
                nil)
```

Within a particular call to `map`, there are two expressions that can be evaluated in parallel: the processing of the current data item, represented by `(f (head lst))`, and the processing of the remainder of the list, represented by `(map f (tail lst))`. This function is compiled in such a way that the second expression will be placed in the spark pool, and then the first will be evaluated. If the maximum number of local service calls are already in progress, then the frame corresponding to `(f (head lst))` will be postponed, since in this case, `f` represents a service call.

Because additional work is only generated when the next `map` call runs, this sparked frame needs to be able to execute while the service requests are in progress. If it did not, then each additional `map` call would only become active once a service call completes, causing it to generate one new service request and one new `map` spark. This would mean that only a single machine could be kept busy. Instead, by allowing the `map` spark to be activated immediately once the asynchronous connection attempt begins, many additional sparks can be generated and handed out to idle machines. The result is that all of the machines receive enough work to achieve an even load balance.

Postponing connection attempts has a major impact on the performance of this program. Figure 6.13 shows the execution time both with and without postponing enabled, for varying numbers of sparks distributed in response to each work request. These results were obtained using the code given above, which invokes 4,096 service requests involving 128 ms of computation each. The solid lines show the average time observed over five runs, and the dashed lines represent the minimum and maximum times. All runs used eight dual-processor nodes.

With postponing disabled, connections are always made on the machines to which their sparks are initially assigned. Because work requests may arrive at unpredictable times, the amount of work that gets assigned to each machine varies greatly, and can be anywhere between none and almost all of the requests. The execution times observed in this case are much higher than those seen with postponing enabled, and exhibit a high degree of variability.

With postponing enabled, sparks that are initially assigned to a particular machine and end up being postponed can be redistributed to other machines that later become idle. Because of this, the amount of work a machine is initially assigned has a much smaller impact on the eventual load balance and hence overall execution time. Machines which become idle simply send out new work requests, and obtain postponed sparks from other machines which are still busy executing requests. The execution times observed in this case are significantly lower than with postponing disabled, and there is much more consistency across multiple runs with the same parameters.

6.4.7 How many frames to distribute?

Looking at Figure 6.13, it can be seen that the amount of time taken to execute the workflow in the case where postponing is enabled varies significantly based on the number of frames distributed in response to each work request. In this graph, which represents an execution involving 128 ms of computation per service call, the best performance is only obtained once the frame distribution count reaches about 32. Below this, the delay between idle machines sending out work requests and receiving a response starts to contribute significantly to the execution time. Distributing more frames at a time reduces the frequency of work requests by enabling machines to stay busy for longer.

The performance differences observed in Figure 6.13 indicate that it is important to pick an appropriate frame distribution count. However, based the above results alone, it is not clear whether 32 frames is necessarily enough for all granularities. If this value is too low, it will result in workflows with fine-grained calls achieving less than their optimal performance. Values that are too large can slightly decrease performance, because machines are given more work at a time than they can reasonably handle, and repeated re-distribution of frames needs to occur more often.

To determine an appropriate default to use for the frame distribution count, we ran an experiment to measure the performance with varying values for different request granularities ranging from 32 to 1,024 ms per call. As in the previous section, each test used

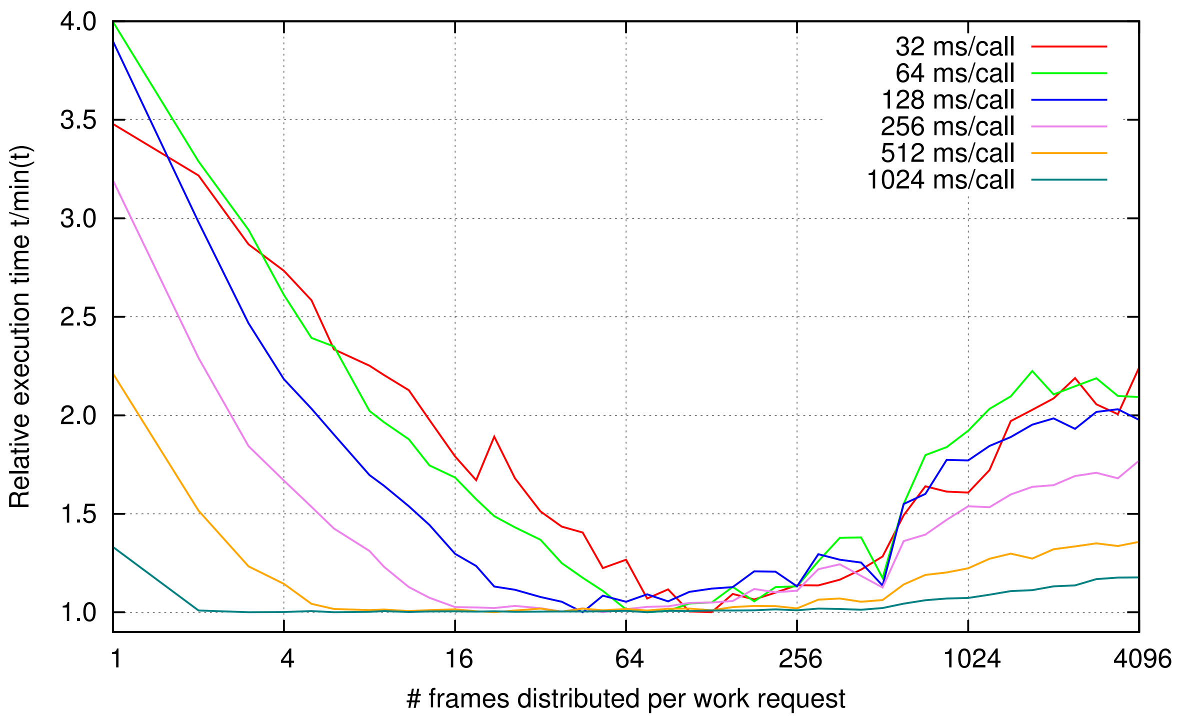


Figure 6.14: Effect of frame distribution count

a total of 4,096 requests. Since the fixed number of requests and varying granularity resulted in vastly different total amounts of computation time, we compared the results of each in terms of the ratio of the execution time to the best average for a given test. Figure 6.14 shows the results, which are the averages taken over five runs.

As expected, the increase in execution time is greatest for workflows containing fine-grained calls. Even with a frame distribution count of 2, the 1,024 ms/call test achieves close to its best performance, but for the 32 ms/call test, the best performance is only observed at around 128. For large distribution counts, each of the tests decreases in performance, with the finer granularity tests seeing greater increases sooner. Based on these results, we concluded that the best choice was a value of 128, which is what we use for all of the experiments in Chapter 7.

In this section, we have looked at very fine-grained service calls, as low as 32 ms each. Many of the operations in typical workflows involve significant amounts of computation, and are thus considerably more coarse-grained than this. The reason for examining fine-grained calls was to establish an upper bound on the performance of our system. Workflows involving large numbers of fine-grained calls require the ability to fire off many requests per second. In the following section, we examine what the limits are on the frequency of requests.

6.5 Limits on connection rates

TCP is designed for applications that use relatively small numbers of long-lived connections. The purpose of the connection abstraction is to provide a bi-directional stream of data over which two hosts may communicate. This is in contrast to UDP, which permits short messages to be sent between hosts without any prior handshaking, but does not provide the reliability, flow control, or streaming capabilities of TCP.

RPC protocols that utilise TCP may either use persistent connections, where multiple requests and responses are sent across the one connection, or alternatively a separate connection for each call. This latter approach is simpler to implement, which is the reason we use this as the model for enabling service access from within a workflow. However, it has the drawback that the rate at which remote procedure calls can be made is limited by the rate at which connections can be established.

In TCP, there is a practical limit on the connection rate between any given pair of hosts, regardless of the speed of the network or individual hosts. This limit arises from the fact that when a connection closes, it is required to remain in the TIME_WAIT state for some period, to ensure that any delayed segments that still exist in the network do not accidently get interpreted as part of a subsequently established connection stream between the same two hosts and port numbers [96]. The convention used is to keep each connection around for twice the maximum segment lifetime (MSL). Most implementations use a MSL value of between 30 seconds and two minutes, meaning that a connection may remain in the TIME_WAIT state for between one and four minutes after both ends have finished with it.

To understand how this issue limits the connection rate, consider the fact that there is an upper limit of 65,536 ports on any given IP address. Connections are defined by a four-element tuple containing the client IP, client port, server IP, and server port. When a client opens a connection to a server, there is a maximum of 65,536 port numbers that it may use. In practice, this limit is usually at most 64,512, because most operating systems reserve the port numbers 0 – 1,023 for use by privileged processes only. The range of port numbers used for outgoing connections, as assigned by the `connect` system call, is usually less than this. For example, the default outgoing port range on Linux is 32,768 – 61,000 — a total of 28,232 ports.

Since the default MSL value on Linux is 30 seconds, each connection that is closed by the client will remain in the TIME_WAIT period for a period of one minute (2×MSL). A client may thus open and close up to 28,232 connections per minute, or 470 per second, before encountering a situation in which there are no more ports available for use. If the client tries to establish connections at a higher rate, then when the ports run out, the `connect` system call will fail with an error indicating that the TCP stack cannot assign it an address.

Of course, 470 new connections per second between a given client and server is many times more than most applications need. In fact, TCP is not really intended to be used in such a manner, and applications that need to send messages at such a rate are better off using another protocol such as UDP, or multiplexing messages across a single connection, to avoid the overheads of TCP connection establishment. Nevertheless, the (admittedly simple) model we have chosen is to make each service call using a separate connection, and in the interests of determining the performance limits of our system, we looked at ways to go beyond this limit.

One approach to increasing the maximum connection rate is to alter the system configuration so that the TCP stack uses a wider range of port numbers for outgoing connections, and/or uses a smaller MSL value. However, this option is only available to privileged users, and given that our testing was performed on a shared cluster, we elected to avoid relying on system-wide configuration changes. However, it is possible for user processes to override both of these restrictions on a per-connection bases, and it is this approach we chose instead.

Normally, a client wishing to establish a connection to a server simply creates a socket and invokes the `connect` system call. The TCP stack chooses an outgoing port to use for the connection from the default range. However, it is possible for a client to manually specify the port to be used by binding the socket to the desired port prior to calling `connect`. If this is done, the TCP stack will instead use the application-specified port number for the connection, assuming there is not already a connection to the same destination IP and port that uses this local client port. If there is, the client must set the *reuse address* option on the socket, to instruct the kernel to remove the existing connection if it is in the TIME_WAIT state, effectively ignoring the MSL value.

This solution technically violates the TCP specification, since hosts are supposed to ensure that closed connections remain in TIME_WAIT for up to twice the maximum segment lifetime. Ignoring this risks the possibility that there will be delayed segments in transit through the network from the previous connection, which may arrive at the receiver after the subsequent connection with the same client IP/port and server IP/port is established. In this situation, the client could potentially receive incorrect data. In general, it is best for applications to respect this requirement of TCP and not reuse TIME_WAIT connections.

We are not suggesting that this approach is desirable in normal situations, since it is better for clients to respect the safety requirements of TCP. However, in order to benchmark the performance of our system and determine the maximum rate at which the virtual machine is capable of handling service requests, we have bypassed this restriction for some of our testing. The benchmark described in Section 7.2.1 shows that it is possible for the virtual machine to invoke up to 1,200 service calls per second on the test hardware.

As mentioned above, it is desirable for applications that require lots of messages/requests to be sent to a server to instead multiplex them over long-lived connections. HTTP's support for persistent connections [102] was designed for exactly this purpose, and in the context of web services is a sensible approach when sending multiple requests to the same server. One area of future work for us is to incorporate the ability to make HTTP requests using a built-in function similar to `connect`, and have the virtual machine handle multiplexing of requests over a single connection per server.

6.6 Summary

This chapter has discussed a number of practical issues that present challenges to maximising the performance of workflows that access external services. We have presented solutions to these issues which we have implemented in our system, and are important for achieving the performance results reported in the next chapter. It is hoped that the discussion presented here may serve useful to other workflow engine developers, as well as those producing other client software which makes use of large numbers of service calls or web requests.

Of the two common evaluation modes used in functional languages, we found strict evaluation to be the most appropriate for our needs. The main reason for this is that it better facilitates automatic parallelisation, in comparison with lazy evaluation. We also found that in the context of the service access model we support, lazy evaluation can cause deadlocks in certain situations where connections are kept open for much longer than necessary, and servers limit the number of connections they are willing to accept.

Clients making large numbers of requests to a server must be careful to exercise restraint, to avoid overloading the server and risking having connections rejected. In doing this however, they should also strive to make use of all the parallelism that is available on the server, particularly in the case of multi-core machines and server farms. Picking a suitable number of requests to allow open at a given point in time requires consideration of how many connections the server(s) at a given IP address are willing to accept, and to distinguish between *outstanding* connections, which reside in a queue, and *accepted* connections, which are being processed by a service. A limit should be placed on the former, but not on the latter.

Load balancing of external service requests is a complicated task in the case of the distributed execution mode used for choreography. The virtual machine nodes must be careful to avoid overloading services, while permitting parts of the program that generate additional parallelism to run. This is achieved by distinguishing between different types of sparks, where those which invoke local services are postponed if a machine is too busy, while others which may generate additional parallelism are still allowed to run. Additionally, the number of frames distributed per work request also has an effect on performance, so it is important to choose an appropriate value.

These are just some of the implementation issues that we have had to address in order to achieve efficient execution of workflows in our system. These types of issues illustrate some of the complexities that arise from scheduling workflows based on a Turing complete programming model, in which the available parallelism cannot be statically determined. We have successfully addressed these issues, and in the next chapter we discuss in detail the results that we have been able to obtain using the system as described in previous chapters and the solutions covered in this chapter.

Chapter 7

Performance Evaluation

This chapter addresses the evaluation of the system we have developed, as described in previous chapters. We seek to determine the degree to which it meets the goals of this project, which are to develop a workflow engine that:

1. Efficiently executes fine-grained computation and data manipulation included directly within a workflow
2. Supports the parallel invocation of multiple external services, without requiring explicit use of any parallel programming constructs
3. Achieves performance gains commensurate with the number of machines and amount of parallelism available
4. Reduces the amount of data transferred between hosts when used for choreography

Each of these aspects are treated individually, in each case using a set of small example workflows which exploit the necessary features. Where appropriate, we have combined features such as data manipulation and parallelism to demonstrate how these can usefully be combined. The aim of this chapter is to show how well the NReduce virtual machine and our XQuery implementation enable programmers to construct complex workflows out of a set of services available via a network.

We begin our experiments by examining basic performance properties relating to service invocation, in terms of task throughput, number of tasks, and task granularity — these experiments give an upper bound on the size and complexity of workflows that can effectively be used with our system. We then look at several different example workflows based on common forms of parallel processing, such as data parallelism, divide and conquer, and pipelining. For each of these types, we examine the speedup possible for compute- and data-intensive workflows, as well as the extent to which choreography can reduce the network communication that occurs between machines, relative to orchestration. In terms of XQuery, we study an example workflow that performs large amounts of computation and data transfer, and examine its performance with respect to both execution time and network communication. Finally, we examine the performance of internal computation in the context of both ELC and XQuery, which is important for workflows containing significant amounts of application logic.

7.1 Experimental setup

All of the experiments described in this chapter were run on *Hydra*, an 82-node cluster hosted at eResearch SA[1]. Each node is a dual-processor Intel Xeon, running at 2.4 GHz, and equipped with 2 Gb of physical memory. All nodes run Linux 2.6.18, and are connected via 100 Megabit Ethernet. The cluster uses Torque [74] as its batch queuing system.

To coordinate the experiments, we developed a series of shell scripts for generation of job submission files, collation of timing results, analysis of log files to extract network usage statistics, and generation of input files to gnuplot [156] to produce graphs. These scripts enabled experiments to be easily repeated, which was useful for testing performance improvements during development.

Most of our experiments use a generic computation service we developed which allows the client to directly control both the amount of computation performed and the volume of data consumed and produced by the service. Requests to this service consist of a numeric value specifying the number of milliseconds worth of computation to perform, followed by arbitrary data, which is not interpreted by the service. Upon receiving a connection, the service immediately sends back a single "." character to indicate acceptance, carries out the specified amount of computation, and then echoes back the supplied input data. The computation is carried out by a loop which repeatedly executes a set of floating point operations in order to keep the CPU busy. At startup, the service performs a calibration process which determines the number of loop iterations required per second of computation, so that the time required to handle a request depends solely on the computation time specified by the client, rather than the speed of the CPU.

For orchestration, we developed a load balancer, which acts as a proxy for TCP connections. It runs on the same machine as the client, as a separate process. Each connection from the client is established across the local network interface to the load balancer process, which opens a second connection to one of the servers, and forwards data between the two connections. The load balancer keeps track of how many connections are open to each server, and when a new connection request arrives, it selects the server which has the lowest number of active connections at that point in time to service the request. In orchestration mode, an experiment described as using n nodes involves n machines running services, plus one additional machine running both the load balancer and the virtual machine. This mode of operation is depicted in Figure 3.1 on page 91.

For choreography, the virtual machine itself performs load balancing, using the mechanisms described in Section 4.7.4. Each machine on which a service is deployed also runs an

[1]http://www.eresearchsa.edu.au/

instance of the virtual machine, and all of these instances interact with each other to manage execution of the workflow. A client program connects to the remote virtual machine instances, sends them bytecode to execute, and receives the output data produced by the workflow. In choreography mode, an experiment described as using n nodes involves n machines running both a service and a virtual machine instance, plus one additional machine running the client program. This mode of operation is depicted in Figure 3.2 on page 92.

7.2 Service invocation

In this section, we examine basic performance properties of the system relating to scalability of workflows. The goal of these experiments is to determine an upper bound on the size of workflows that our system can support. All of the experiments presented here use the generic computation service described in the previous section.

7.2.1 Number of tasks and task throughput

An important aspect of a workflow engine's performance is how well it scales with respect to the number of tasks contained in the workflow. Each task has an associated memory cost, as it must be represented as an object within some data structure. The scheduling algorithm has an associated time cost, because it must examine the data structure containing the tasks to select which one to execute next. In this section, we measure how well NReduce scales to large numbers of tasks, as well as its maximum achievable *throughput*, which is the rate at which it is able to dispatch tasks.

The simplest way to process a collection of independent tasks in ELC is to use `map` to apply a service invocation function to each value in a list. This enables data parallel processing, because each service call is independent of the others, so all of them can potentially run in parallel, given a sufficient number of processors. Section 7.3.1 measures the scalability of this type of workflow with respect to the number of nodes, and includes the code for the workflow that we used for this experiment.

To measure the maximum throughput, we used only a single machine, since our focus here was on the speed of the task dispatching mechanism; hosting the client and server on the same machine eliminated the influence of network latency. Each request specified zero seconds of computation, and contained no additional input data. The time required to invoke a task was therefore determined by the costs of dispatching the task within NReduce, establishing a TCP connection across the local network interface, and creating a thread within the server process to handle the request.

Figure 7.1 plots the task throughput against the number of tasks in the workflow for up to two million tasks. The results shown are the averages taken over ten runs. For workflows containing up to 1,024 tasks, the results are skewed by the short measurement times, and the startup time of the virtual machine, which is mostly spent performing bytecode

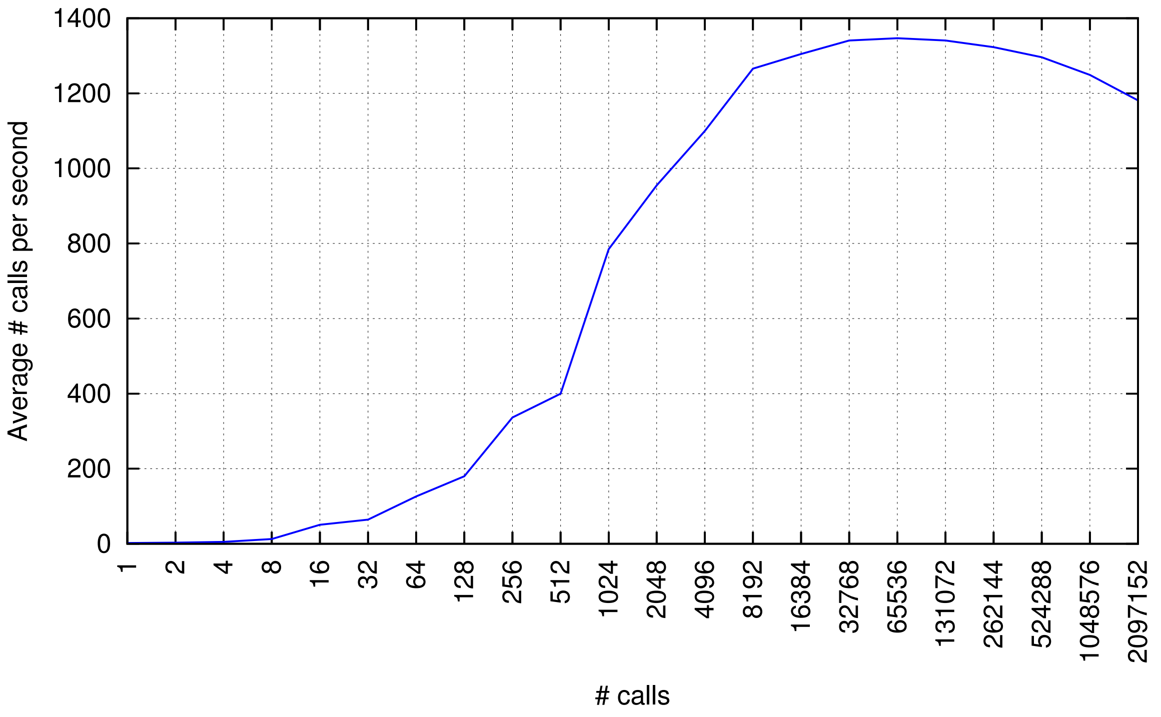


Figure 7.1: Task throughput

compilation of the workflow. Beyond this level, a very high rate of task dispatching is achieved, with over 1,200 calls a second for workflows containing up to one million tasks. In most practical usage scenarios, we anticipate the throughput rate required for most workflows to be many times less than this, so we believe that the throughput achieved here is more than adequate.

The scalability with respect to the number of tasks can be seen by examining how the throughput changes as the number of tasks increases. If scalability were unlimited, the throughput would remain constant. As can be seen however, it starts to decrease slightly after about 65,536 tasks. This decrease is due to garbage collection — more tasks means that more time must be spent performing garbage collection, which slows down the task dispatching mechanism. However, the influence of garbage collection remains small for all but the largest of workflows, and even with two million tasks it only reduces the throughput by around 20%.

For this workflow, the per-task memory cost is just under one kilobyte, so a workflow containing two million tasks requires nearly two gigabytes of memory, which is all that the machines used for the test are physically equipped with. Beyond this limit, the heap spills into swap space. When this happens, garbage collection becomes prohibitively expensive, as it causes the machine to spend a very large amount of time paging data to and from disk, preventing the workflow from completing in a reasonable time. It is for this reason that results for sizes above this amount are not shown.

The maximum workflow size we can support on these machines is around two million tasks. However, that the amount of memory required per service call depends on several factors, including the size of the request and response data, and other code that is used to perform internal computation within the workflow. For this reason, the maximum number of tasks for some workflows may be substantially less than shown here.

7.2.2 Granularity

An important characteristic of a workflow engine is the granularity of tasks that it is able to support, while still delivering good speedup in the context of parallel execution. There is always a certain amount of overhead per task, relating to factors such as scheduling costs and task life cycle management. Large overheads imply that the workflow engine is only appropriate for workflows containing coarse-grained operations. Small overheads imply that fine-grained operations can be used without the overheads dominating execution time.

In large-scale scientific applications executed on HPC resources, the amount of computation work in each job typically ranges from a few minutes to a few hours. Most batch queuing systems and workflow engines are designed with these types of tasks in mind. In our case, we aimed to support tasks in the order of only a few seconds each. Any computation whose execution time is less than this is best expressed as a function to be executed internally within the workflow itself.

To determine this aspect of performance, we ran an experiment designed to answer the following question: For a fixed total amount of computation, divided among multiple service calls, how fine-grained can the individual calls become before we see a significant increase in total execution time? We expected that for very fine-grained calls, the execution times would be high, due to the overheads involved with invoking a large number of service calls involving only a small amount of computation each. The question was how fine-grained we could make the tasks before the execution time is significantly affected.

In this experiment, decreasing the granularity implies increasing the number of service calls, so the formula (amount of computation per call) × (number of calls) gives the same total amount of computation. This means that the performance is also influenced by the extent to which the virtual machine is able to handle large numbers of tasks, which was investigated in the previous experiment. As was shown there, the number of calls can scale to at least a few hundred thousand before performance degradation occurs, provided that the amount of memory used per call remains fairly small.

We ran this experiment using the data parallel workflow described in Section 7.3.1, which invokes the generic computation service using TCP connections from within ELC. The tasks were executed in parallel across eight dual-processor nodes. The total amount of computation time across all service calls was fixed at 65,536 ms, with the granularity of each call ranging from 4 to 2,048 ms. Thus, at the finest granularity, 16,384 service calls were made, each performing 4 ms of computation. At the coarsest granularity, 32 service calls were made, each performing 2,048 ms of computation. Figure 7.2 shows the results, which are the averages of three runs of each test.

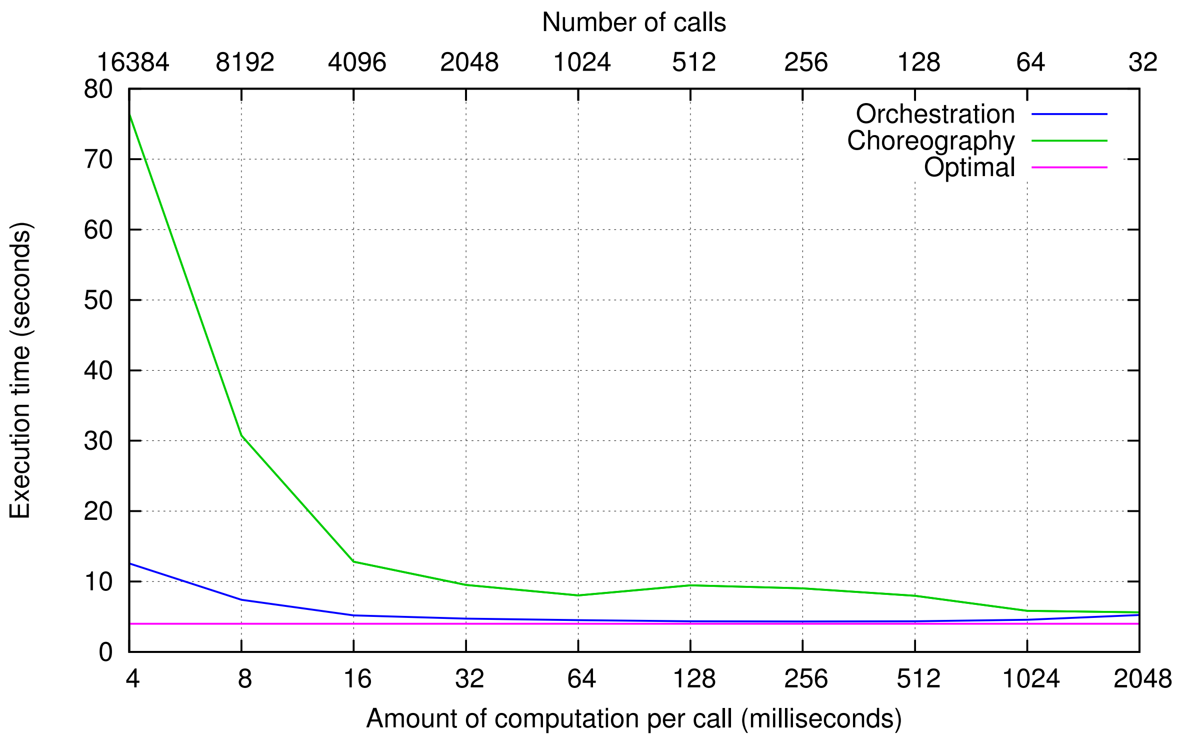


Figure 7.2: Granularity - Direct TCP service calls from ELC

As these results show, the execution time increases sharply below 32 ms, indicating that the overheads of invoking service calls start to become a major component of the execution time. This increase is much greater for choreography, which carries larger overheads for scheduling work to machines in comparison with orchestration. Above 32 ms, orchestration achieves near-optimal performance, taking around 4.5 seconds on average, which is close to the optimal time of 4 seconds. There is a slight increase at 2,048 ms, but this is due to imperfect load balance with a small number of calls, rather than issues of granularity. The execution time for choreography changes gradually above 32 ms, reaching near-optimal performance only at 1,024 ms and above. We attribute this to the latency between sending out a work request and receiving a response. Since an idle machine has no information about which of its peers have sparks available, the work request is sent to a randomly-selected machine, and may need to travel between several different machines before it ends up on one that has sparks available.

We also carried out another experiment which addresses the case of web services invoked from within XQuery. The reason for carrying out this additional experiment was that web services involve overheads associated with XML parsing and serialisation, which we expected would have a significant effect on performance, particularly for fine-grained calls. In the last experiment, all the requests and responses were numbers, which require only negligible time to convert to and from strings.

Figure 7.3 shows the results of the web services experiment, which used the same parameters as above. As can be seen from this graph, the execution time for large numbers of fine-grained calls is several times more than in the previous experiment (note the different

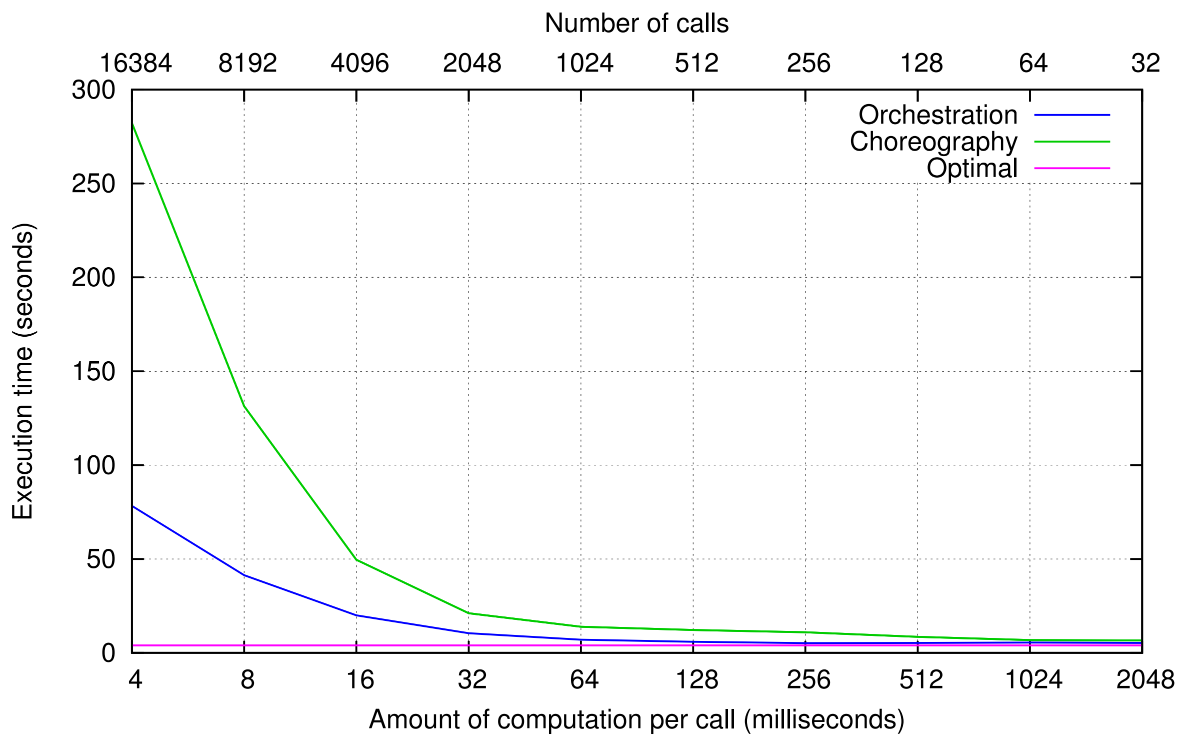


Figure 7.3: Granularity - Web service calls from XQuery

Computation time	Number of calls	Orchestration (s)		Choreography (s)	
		TCP service	Web service	TCP service	Web service
4	16384	12.6	78.3	76.4	282.2
8	8192	7.4	41.4	30.7	131.4
16	4096	5.2	20.0	12.8	49.6
32	2048	4.7	10.5	9.5	21.1
64	1024	4.5	7.0	8.0	13.9
128	512	4.4	5.9	9.5	12.2
256	256	4.3	5.2	9.0	10.9
512	128	4.4	5.2	8.0	8.5
1024	64	4.6	5.5	5.8	6.8
2048	32	5.2	5.3	5.6	6.6

Table 7.1: Granularity comparison (optimal time is 4 seconds)

scales of the two graphs). This difference can be accounted for by the time required to generate the SOAP requests and parse the SOAP responses, both of which are encoded in XML. Both the parsing and serialisation logic are implemented in ELC, so the execution speed of NReduce is a factor in these results. Once the call granularity reaches 512 ms, the difference in execution times becomes small. This can be seen more clearly in Table 7.1, which presents the results from the two experiments side-by-side.

7.3 Types of parallelism

In this section, we measure the speedup achievable for a number of different types of parallelism. The goals of this section are to demonstrate how well-known types of parallel processing can be expressed in ELC, and to determine how well the parallel execution facilities of the virtual machine are able to exploit these workflow structures. As we shall show, the speedups obtained are close to what one would reasonably expect in each case, given the amount of parallelism available in the workflow. The results we present illustrate that our approach to detecting and managing parallelism works well for a range of different workflow structures.

- **Data parallelism** is where the same operation is performed on every item in a data structure in parallel. In some forms of data parallelism, all of the operations can execute independently of each other, such as when using `map` to apply a function to every item in a list. We consider this embarrassingly parallel style in both a single dimension and in multiple dimensions. In other forms of data parallelism, there can be dependencies between the operations. We consider an example of this in which there are multiple stages of embarrassingly parallel processing, separated by collective processing tasks which operate on the data structure as a whole, and must execute sequentially. These are limited by the sequential component, with the maximum possible performance predicted by Amdahl's law [12].

- **Pipeline processing** is where a sequence of service operations is invoked for each individual item in a list, much like a production line. For workflows based on this structure, we consider two ways of exploiting parallelism which we refer to as *physical pipelining* and *logical pipelining*. Physical pipelining is where each service is tied to a specific machine, and parallelism is achieved by having multiple items undergoing different processing stages at the same time. Logical pipelining is where any service can run on any machine, and multiple items may be undergoing either the same or different stages of processing at a given point in time.

- **Divide and conquer** is a problem-solving technique which works by recursively dividing a large problem into many smaller pieces, and then recursively combining those pieces to product a final result. The actual problem solving occurs in one or both of the divide and combine operations. Both are open to parallel execution, since individual calls to these operations at the same depth of recursion are independent of each other. Examples of divide and conquer algorithms include merge sort and quick sort.

For each type of parallelism, we created a simple workflow which follows the basic algorithmic structure in question, and invokes a number of calls to the generic computation service described in Section 7.1. Using this service enabled us to easily specify the amount of computation that should be performed by each service call, so appropriate values could be chosen for the tests. Since this service echoes back all of the input data that was supplied to it, we were also able to control the amount of data transferred by supplying strings of a particular size as input to the service. Each of the experiments in this section uses a range of different node counts, and the results presented are the averages over three runs.

All of the types of parallelism described above are supported by many existing parallel programming languages. For each of the experiments discussed in the following sections, we expected that the speedups obtained by our system should be close to the best that could be theoretically achieved. For some types of parallelism, near-perfect speedup is theoretically achievable, while for others, the available parallelism is inherently limited by the structure of the problem, which in turn limits the maximum achievable speedup.

We also expected the amount of data exchanged with services to affect the speedup, particularly with orchestration. For a given workflow, the total amount of data consumed and produced by services is independent of the number of nodes on which those services are run. For orchestration, the total data transfer is constant, though for choreography, it may potentially be lower, due to less communication being required between machines.

For each of the experiments, the questions we sought to answer regarding our system are as follows:

- Is all available parallelism detected?
- Does the dynamic scheduling mechanism effectively exploit all of this parallelism?
- Are the speedups achieved close to the maximum that could be expected for the type of parallelism in question?

Each of the following sections describes two experiments which address a particular type of parallelism but differ in the amount of computation and data exchange involved. The first experiment measures speedup for a compute-intensive workflow, which uses short service calls and does not send or receive any data except for a single number used to specify the computation time for the service. The second experiment measures speedup for a workflow which is both compute- and data-intensive, involving data transfer in the order of several hundred megabytes in total. In all cases, the second experiment also contains a larger amount of computation than the first, so that it is still possible to achieve reasonable speedups, since the transfer of data to and from services contributes to the total execution time, negatively impacting speedup. The purpose of these data-intensive experiments is to determine the degree of overheads introduced by our system that are in addition to the cost of transferring data across the network.

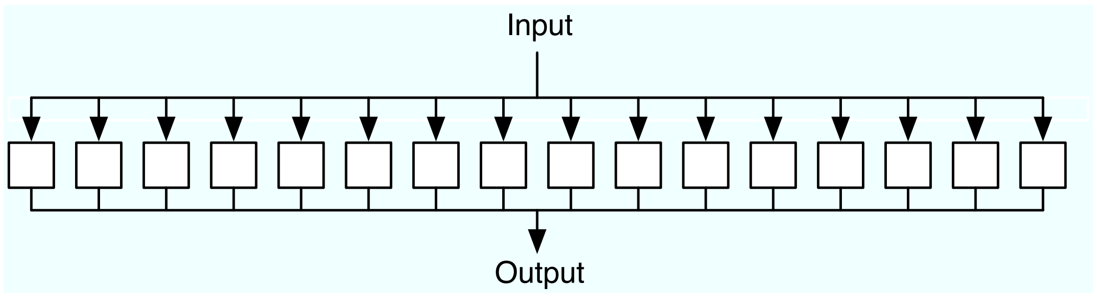


Figure 7.4: Data parallelism

7.3.1 Data parallelism

Data parallelism involves performing the same operation on multiple elements of a data structure in parallel. In this section and the next, we consider the simple case in which each operation is independent of the others, which makes the processing embarrassingly parallel. This model can theoretically result in near-perfect speedup, provided the number of list items is large enough to keep all processors busy, all operations take the same amount of time, and the time required to transfer the input and output to and from the processors is sufficiently small relative to the computation time. In the context of workflows, data parallelism corresponds to invoking multiple calls to a given service in parallel, providing a separate item from a data structure as input to each call. Figure 7.4 depicts data parallel processing of a list, where service calls are represented as boxes, and lines and arrows indicate data dependencies.

In ELC, one way of expressing data parallel processing is to use `map`, which applies a function to every value in a list. As discussed in Section 6.4.6, `map` is implemented by traversing through the list, constructing a new cons cell at each step. The head of the cons cell is the result of applying the specified function to the current item, and the tail is the result of applying `map` to the remainder of the list. With strict evaluation, both arguments to `cons` must be evaluated before the cons cell can be constructed; this recursively triggers parallel evaluation of the entire list, generating a spark for each item. When a workflow contains a call to `map` and supplies a function which invokes a service, this causes the service to be invoked for each item in the list, and these invocations can potentially occur in parallel.

We tested data parallelism by constructing a simple workflow which produces a list of items of a specified size, and invokes the generic computation service for each of them. The code used for this workflow is shown in Figure 7.5. It takes command-line arguments specifying the number of items in the list, the amount of computation to request the service to perform on each, and the number of bytes of data to send in each request. It then creates a list containing the specified number of copies of a string of the specified size, and uses `map` to invoke the `callservice` function for each. The result is a string containing the data echoed back by all of the service calls.

Because of the use of `map`, we can expect that at the beginning of execution, sparks for each item in the list should be generated and placed in the spark pool, making the

```
main args =                                 // Main function
(if (< (len args) 5)                        // Check command-line args
  "Usage: dataparallel.elc host port nitems compms itemsize\n"
(letrec
  host = (item 0 args)                      // Process command-line args,
  port = (ston (item 1 args))               // using ston to convert strings
  nitems = (ston (item 2 args))             // to numbers for numeric args
  compms = (ston (item 3 args))
  itemsize = (ston (item 4 args))

  input = (genstring itemsize)              // Generate input string of
                                            // requested size

  inputlist = (map (!n.input)               // Create input list with n
                   (range 1 nitems))        // copies of input string

  callservice value =                       // Service stub function: connect
    (connect host port                      // to service and send request
    (++ (ntos compms) (++ " " value)))      // containing comp time & data

  results = (map callservice inputlist)     // Invoke service for each item
                                            // in input list
in
  (foldr ++ "\nEND\n" results)))            // Print result list, terminated
                                            // by "END" to confirm completion
```

Figure 7.5: Workflow for testing data parallelism

maximum amount of parallelism available. The dynamic scheduling mechanism should repeatedly consult the spark pool whenever there are one or more machines idle. Because all of the parallelism is available from the beginning, and there are enough sparks to keep all of the processors busy, we should expect to see near-perfect speedup.

Both of the experiments for this workflow used a list size of 1,024 items. The first involved a computation time of one second per service call, and an item size of zero. The second involved a computation time of five seconds per service call, and an item size of 256 Kb[2]. The second experiment thus involved 256 Mb of data being sent to the services and echoed back, for a total of 512 Mb[3]. The results from these experiments are shown in Figure 7.6.

For the first experiment, the speedup for orchestration was close to perfect. The speedup for choreography was slightly lower, which we attribute to overheads involved with the load balancing mechanism used by the virtual machine. In this mechanism, idle nodes

[2]Kb means *kilobytes*, where one kilobyte is $2^{10} = 1{,}024$ bytes.

[3]Likewise, Mb means *megabytes*, where one megabyte is $2^{20} = 1{,}048{,}576$ bytes.

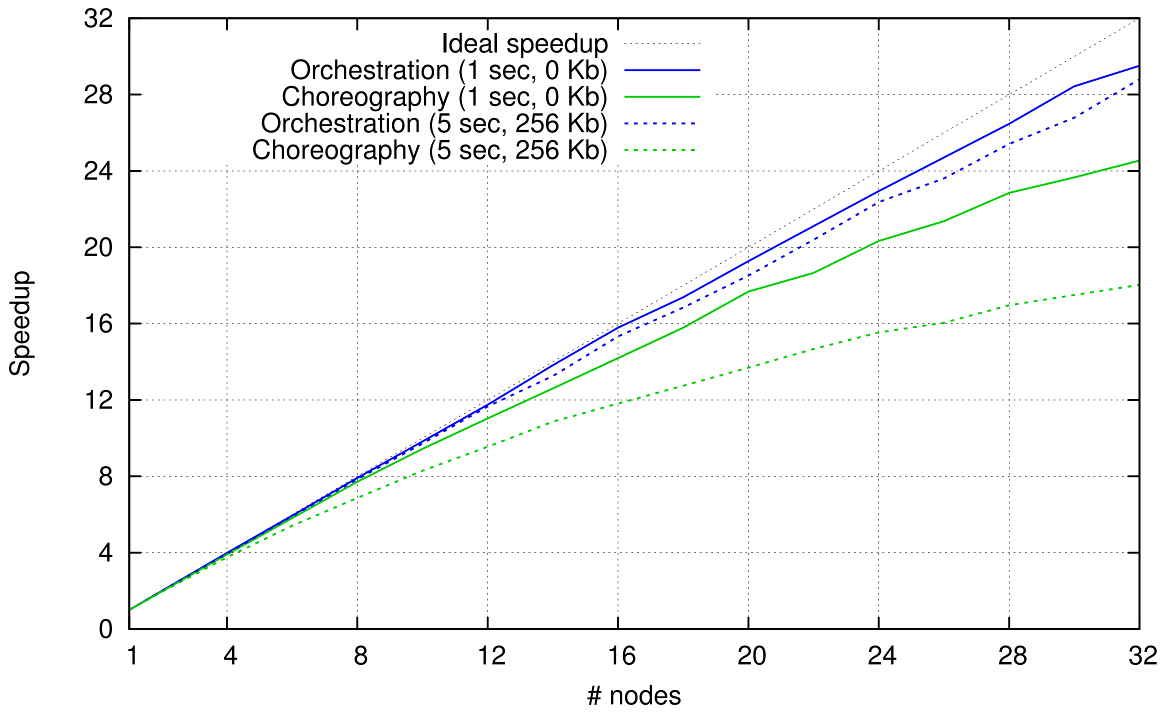


Figure 7.6: Data parallelism — speedup (1,024 items)

communicate with their peers to search for work that they can perform, as described in Section 4.7.4. This involves slightly higher delays than the load balancer used for orchestration, which maintains local state indicating which machines are busy and can thus make an immediate decision about where to send a request. For choreography, the work request may have to visit a number of different machines before finding one that has sparks available, so there is a delay between when an idle node sends out a work request, and when it receives sparks that it can execute.

For the second experiment, both orchestration and choreography achieved lower speedups, with a greater reduction seen for choreography. This experiment involves 5 × 1,024 = 5,120 seconds of computation, so for 32 nodes (64 processors), perfect speedup would correspond to an execution time of exactly 80 seconds. For orchestration, the observed results are close to this — around 90 seconds on average. The additional ten seconds can be accounted for by communication costs. Mostly, the exchange of input and output data with the services overlaps with the computation, so the majority of the 512 Mb exchanged with the services is transferred while other calls are in progress. However, this overlap is not complete, because in some cases there are short delays while a result is transferred back before the server allows the next connection to be opened, and at the end of execution, the final calls need to have their results returned to the client.

For choreography, the same amount of data is transferred over the network, but the total execution time is much longer — 147 seconds on average for 32 nodes. In this case, almost none of the communication overlaps with computation. When each service call completes, the result is stored on the virtual machine node that invoked it. Only at the

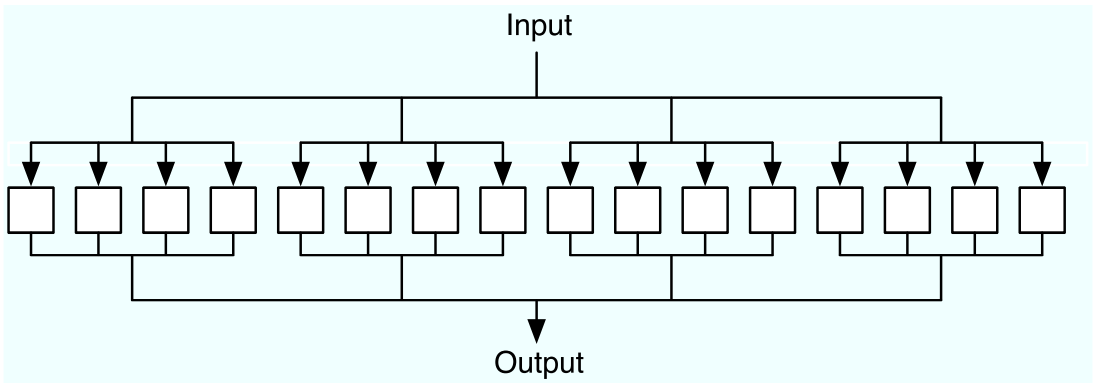


Figure 7.7: Nested data parallelism

end of execution, when all service calls have completed and their results are collated into a single output string, are these results transferred across the network. When this occurs, one of the virtual machine nodes executes the collation process, carried out by the `foldr` expression at the bottom of Figure 7.5.

Due to strict evaluation, the complete list returned by `map` must be fully evaluated (meaning all the service calls must be made) before the initial call to `foldr` can begin. Once this has occurred, `foldr` traverses the list sequentially, and for each item, fetches its contents from the node that holds the corresponding service result. Again, due to strict evaluation, the runtime system does not begin writing the results of `foldr` to the client's output stream until the complete program output has been constructed. Together, these two final phases result in the transfer of 512 Mb of data, which takes 147 - 80 = 67 seconds. This corresponds to a transfer rate of 7.6 Mb/second, which is close to what would be expected for the 100 Megabit network connecting the nodes.

Of course, it would be preferable if there were some way to make this communication overlap with the computation, so that each item in the list returned by `map` is immediately sent back to the node which will subsequently consume it. However, the runtime system cannot tell for certain which node this will be, since it is possible that the consuming expression could separately be scheduled to another node. Given our current evaluation mechanism, it is not immediately obvious how this problem could be solved, but it is certainly worthy of investigation in future work. This outcome demonstrates that in some cases, orchestration is clearly preferable, especially given that in this particular situation there is no exchange of data between services, so choreography would not offer benefits for this workflow even if this problem did not exist.

7.3.2 Nested data parallelism

Nested data parallelism is where each item in a multi-dimensional array is processed in a data parallel fashion. It is a more general case of the flat data parallelism studied in the previous section, which is only one-dimensional. For the nested case, we studied a two-dimensional array of values, i.e. a list of lists. Each element in the array corresponds to a

```
main args =                           // Main function
(if (< (len args) 5)                  // Check command-line args
  "Usage: nested.elc host port size compms itemsize\n"
(letrec
  host = (item 0 args)                // Process command-line args,
  port = (ston (item 1 args))         // using ston to convert strings
  size = (ston (item 2 args))         // to numbers for numeric args
  compms = (ston (item 3 args))
  itemsize = (ston (item 4 args))

  callservice n =                     // Service stub function -
    (connect host port                // connect to specified server
    (mkrequest compms n))             // and send request

  input = (genstring itemsize)        // Generate array element content
                                      // of requested size

  array = (create2darray size input) // Construct size*size input array

  result =
    (map (!row.map callservice row)  // For each column in each row,
         array)                       // call the service
in
  (++ (str2darray result)             // Print output array, terminated
      "END\n")))                      // by "END" to confirm completion



str2darray array =                    // Array to string: Loop over rows
(foldr ++ ""                          // and columns, concatenating column
       (map (!row.foldr ++ "\n" row) // elements on a single line, and
            array))                   // row elements on separate lines

create2darray size content =          // Create input array:
(map (!row.map (!col.++ (ntos row)   // For each row and column, create
                    (++ " "           // array element containing row num,
                    (++ (ntos col)    // column num, and input string
                    (++ " " content))))
               (range 1 size))
     (range 1 size))

mkrequest ms data =                   // Build request:
(++ (ntos ms)                         // Send no. ms of computation, and
(++ " " data))                        // input data to be echoed back
```

Figure 7.8: Workflow for testing nested data parallelism

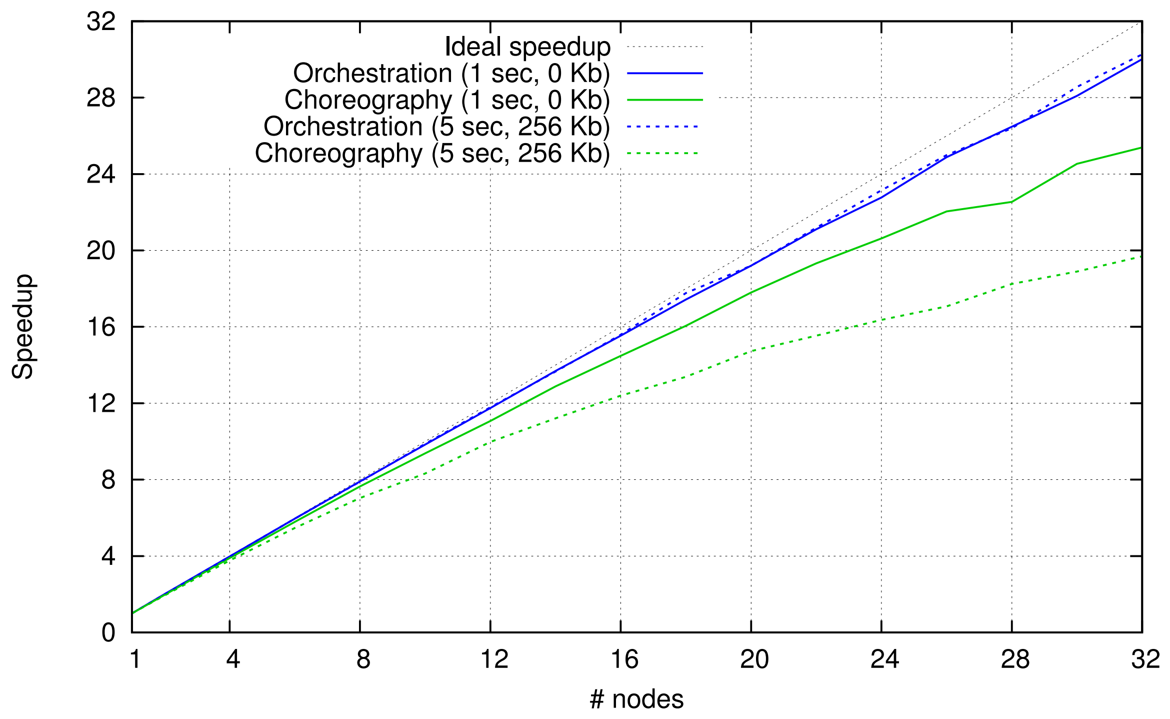


Figure 7.9: Nested data parallelism — speedup (32 × 32 items, 1 s computation per call)

string which is sent as the input data to the generic computation service. This structure is shown in Figure 7.7.

The workflow used for the experiments in this section is shown in Figure 7.8. It takes command-line parameters specifying the size (number of rows and columns) of the array, the amount of computation per service call, and the size in bytes of each array value. This workflow first creates a two-dimensional array, each element of which contains the row number, column number, and input data of the specified size. The expression bound to `result` iterates over this array using two nested calls to `map`, with the inner loop body invoking the `callservice` function for each item. This expression produces another two-dimensional array containing the data echoed back by the service, which is then serialised to a string using the `str2darray` function. This string forms the value returned by the `main` function, which is printed to standard output.

The parameters used for both experiments conducted with this workflow were the same as those from the previous section, except that the list size was set to 32, so that in total the two-dimensional array contained 32 × 32 = 1,024 elements. As before, the first experiment involved one second of computation per service call, and zero-byte input strings. The second experiment involved five seconds of computation per service call, and 256 Kb of input data per item, for a total of 512 Mb exchanged with the services. In both cases, the results were expected to be identical to those from the experiments described in the previous section, because the experiments involve the same amount of computation and data transfer, and generate the same number of sparks at the beginning of execution. The results are shown in Figure 7.9.

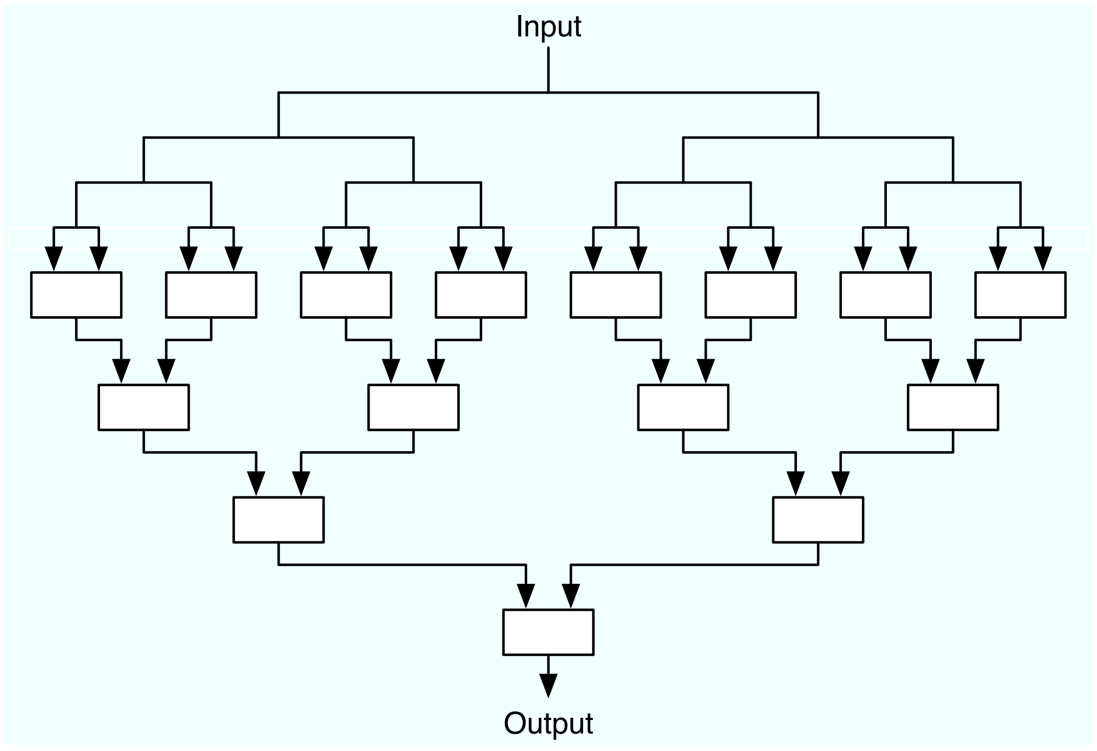


Figure 7.10: Divide and conquer parallelism

As with the one-dimensional data parallel experiments, orchestration and choreography achieved close to the ideal speedup for the first experiment, with choreography achieving slightly lower speedup due to work distribution overheads. In the second experiment, the results for orchestration were just as good for the first, but choreography was significantly slower for the reasons discussed previously, relating to transfer of data after the service calls had been completed.

The fact that these results are virtually identical to those presented in the previous section illustrates that our generic approach to detecting parallelism is agnostic regarding the structure in use. This point will become clearer as we discuss further experiments involving other forms of computation in the following sections. The speedup on these experiments depends on the amount of available parallelism, not the way in which it has arranged. If we had tried to cater for nested data parallelism as a special case, it is possible we may have observed different results for these experiments. It can be expected that with our implementation, flat and nested data parallelism will perform identically in all cases.

7.3.3 Divide and conquer

Another type of algorithmic structure amenable to parallel execution is *divide and conquer*. Algorithms of this form start out with a large initial problem, and divide it into two or more smaller instances of the same problem. These are solved recursively, and then the solutions are combined to produce a solution to the initial problem as a whole. This

```
main args =                                 // Main function
(if (< (len args) 5)                        // Check command-line args
  "Usage: divconq.elc host port nitems compms itemsize\n"
(letrec
  host = (item 0 args)                      // Process command-line args,
  port = (ston (item 1 args))               // using ston to convert strings
  nitems = (ston (item 2 args))             // to numbers for numeric args
  compms = (ston (item 3 args))
  itemsize = (ston (item 4 args))

  input = (genstring itemsize)              // Generate input string of
                                            // requested size

  inputlist = (map (!n.input)               // Create input list with n
                   (range 1 nitems))        // copies of input string

  callservice x y =                         // Service stub function -
   (connect host port                       // concatenate args and
            (++ (ntos compms)               // pass to service
            (++ " " (++ x y))))

  result = (divconq callservice             // Apply divide and conquer using
                    inputlist)              // callservice as the combining
                                            // function
in
  (++ result                                // Print result, terminated by
      "\nEND\n")))                          // "END" to confirm completion

divconq f lst =                             // Divide and conquer function
                                            // (f is the combining function)

(if (== (len lst) 1)                        // Single item: return it
    (head lst)
(if (== (len lst) 2)                        // Exactly two items:
    (f (item 0 lst)                         // combine them
       (item 1 lst))
(letrec                                     // More than two items:
  mid = (floor (/ (len lst) 2))             // Split list in half
  right = (- (len lst) mid)
in                                          // Solve recursively, and
  (f (divconq f (sub 0 mid lst))            // combine results
     (divconq f (sub mid right lst))))))
```

Figure 7.11: Workflow for testing divide and conquer parallelism

structure is shown in Figure 7.10. Parallel execution is possible whenever the sub-problems can be solved independently.

In this section, we consider an example workflow in which the division is implemented internally by splitting a list of input values in half, and the problem solving occurs using an external combine function provided by a service. Figure 7.11 shows the code for the workflow. In this case, the input values are strings of a specified size, and the combine calls invoke the generic computation service, which simply echoes back its input. The number of items, amount of computation, and size of input values are specified using command-line parameters.

Both of the experiments we ran with this workflow used an input list with 512 items. The first experiment carried out one second of computation per service call, with zero-byte list items. The second experiment carried out five seconds of computation per service call, with 64 Kb list items. With divide and conquer workflows, choreography opens up the possibility of reducing the amount of data that must be transferred between machines — we examine this in Section 7.4.1.

Figure 7.12 shows the results of both experiments. Orchestration and choreography achieve very similar speedup, with choreography in both cases being slightly slower, due to the scheduling overheads discussed previously. The speedup observed on these experiments is much lower than that obtained for the data parallel tests in the preceding sections. However, this is due to the nature of divide and conquer, rather than our implementation.

Algorithms which follow this structure carry an inherent limitation regarding the amount of parallelism they contain. With a sufficiently large list, the early stages of combining are able to make use of many processors, because there are many pairs of items to be combined. However, as the program proceeds towards smaller and smaller lists, there are fewer calls to the combine function that need to be made. Right at the end of the computation, there remains only a single call, which combines the results obtained for the two halves of the input. It is in these latter stages of execution that the available parallelism significantly decreases.

This behaviour is clearly visible in Figure 7.13, which shows the average processor utilisation for an additional set of experiments we ran involving different list sizes, using ten-second service calls and zero-byte list values, with 32 nodes. As can be seen from the figure, none of the runs fully utilises the nodes for the whole time. In all cases, the highest utilisation is achieved at the start of execution, when the number of items being processed is largest. As each stage of combining completes, the number of items to process halves. This produces the step-like effect that is noticeable for the 32, 64, and 128 item tests. For the remaining two cases, the fact that there are more calls to be made results in increased overlapping of stages, so the individual stages complete more gradually.

As each stage ends, the halving of the list size means that there is less parallelism (fewer sparks) available for the processors to exploit. As long as the number of sparks remains greater than or equal to the number of processors, full utilisation is achieved. Once the number of sparks drops below this threshold however, some of the processors necessarily remain idle, so the average processor utilisation drops below 100%. For the 32 and 64

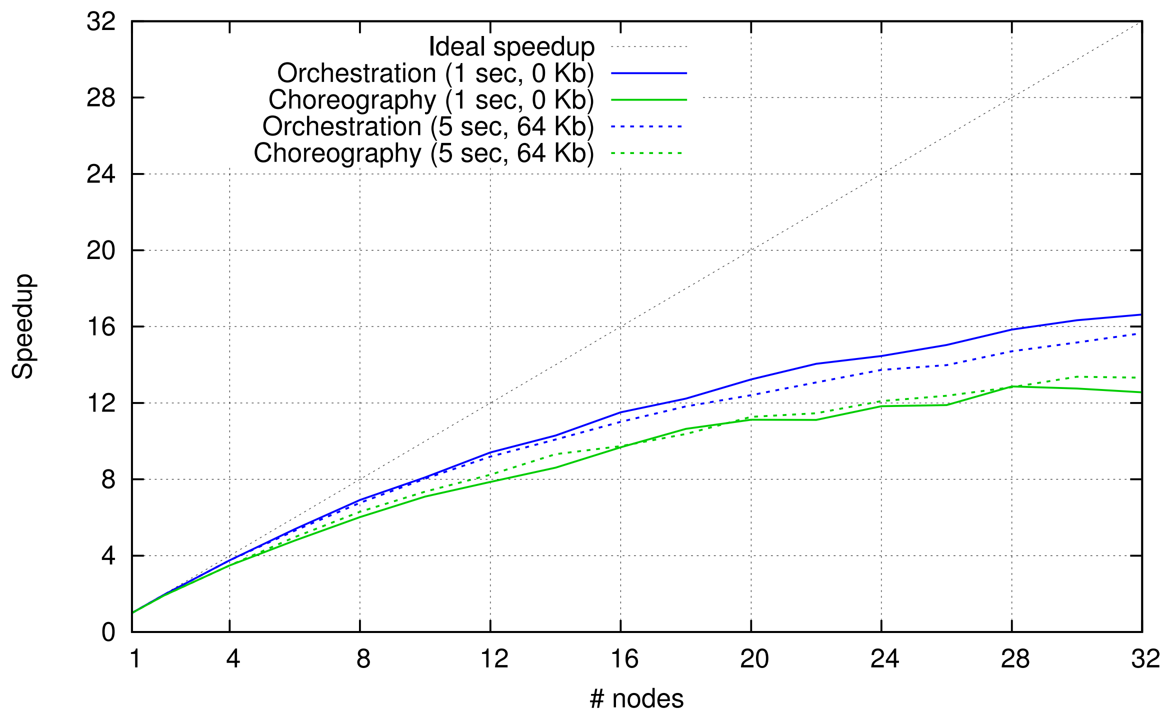


Figure 7.12: Divide and conquer — speedup (512 items)

item tests, there is not even sufficient parallelism in the initial stage of processing to keep all machines busy, which is why these never achieve full utilisation.

Of course, these limitations apply to any implementation of the divide and conquer structure, since they are inherent in the nature of the problem. What this experiment shows, however, is that our implementation is able to detect all of the parallelism that *is* available, and schedule the service calls appropriately. Importantly, the parallelism is detected using the generic sparking mechanism of the virtual machine; we do not need to provide explicit language support for divide and conquer algorithms — they can simply be expressed in user code, as in Figure 7.11.

In this case, parallelism occurs as follows: for branch nodes in the call tree, `divconq` sparks evaluation of both arguments to the combine function, and then blocks waiting for them to complete. These arguments are recursive calls to `divconq`, and when they execute, they will spark further arguments, and block. Very early on during execution, the program will reach a state in which all of the branch calls are blocked, and the leaf nodes are sparked. Since the leaf nodes simply return their arguments, the lowest branches in the call tree will soon become unblocked, and all attempt to invoke the combine function with the results of the leaf calls, causing the lowest-level combine calls to be sparked.

As this process continues, the lower branches of the call tree complete, unblocking the branches above them, which in turn spark further calls to the combine function. However, since each level of the call tree contains fewer calls, there are fewer sparks generated, and thus less work to be scheduled, leading to lower average processor utilisation. The

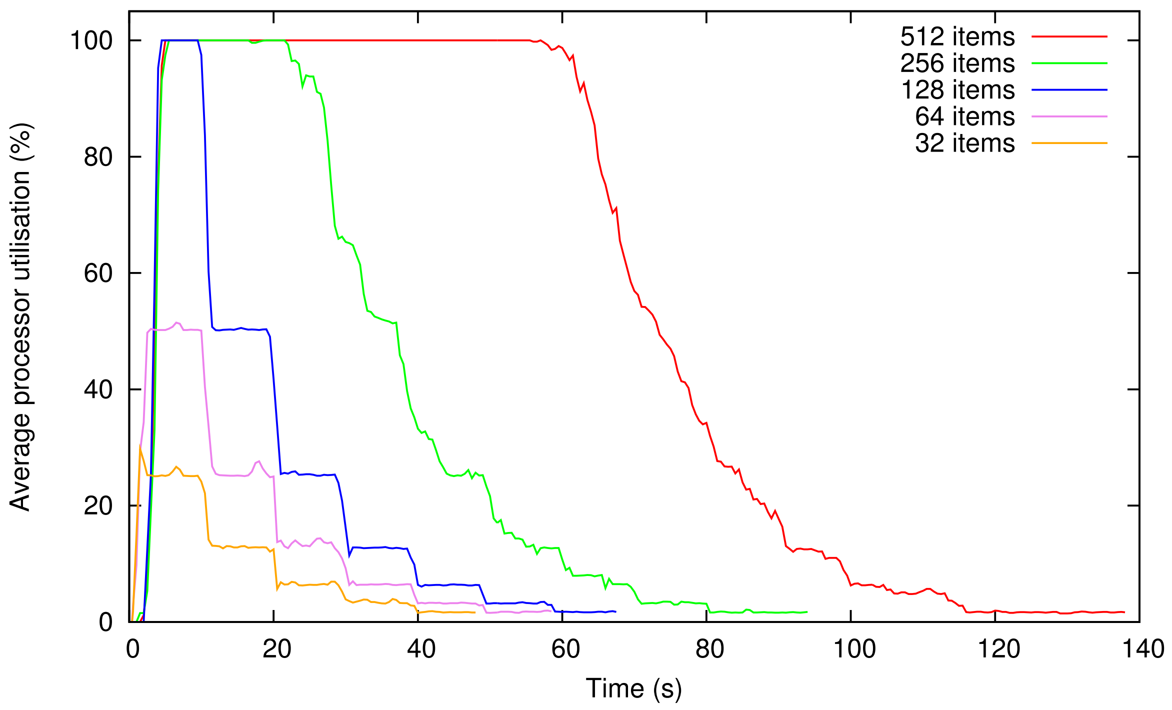


Figure 7.13: Divide and conquer — average processor utilisation (32 nodes)

speedups shown in Figure 7.12 are in line with what one would expect to see in the best case for a divide and conquer algorithm.

7.3.4 Physical pipelining

Many workflows are based on a *pipeline processing* structure, in which each item in a list of values goes through several different processing stages. We refer to the standard notion of pipeline parallelism, in which each processing stage can only run on a specific machine, as *physical pipelining*. In contrast, if each stage can occur on any available machine, we refer to this as *logical pipelining*, which is discussed in the next section.

Physical pipelining must be used when each service is deployed on a separate machine, and the user has no ability to change this deployment arrangement. In a typical usage scenario, there would be a set of publicly accessible services provided by different sites on the Internet, and the workflow would make a call to each of these services for every item in a list. Parallelism can be achieved in this scenario by having all of the services in use at the same time, each processing a different item from the list.

The basic structure of a workflow that uses pipeline processing is shown in Figure 7.14, which depicts the list of items horizontally, and the processing stages each item goes through vertically. Pipeline parallelism exists in the vertical dimension, where items at different stages can be processed in parallel. Data parallelism exists in the horizontal

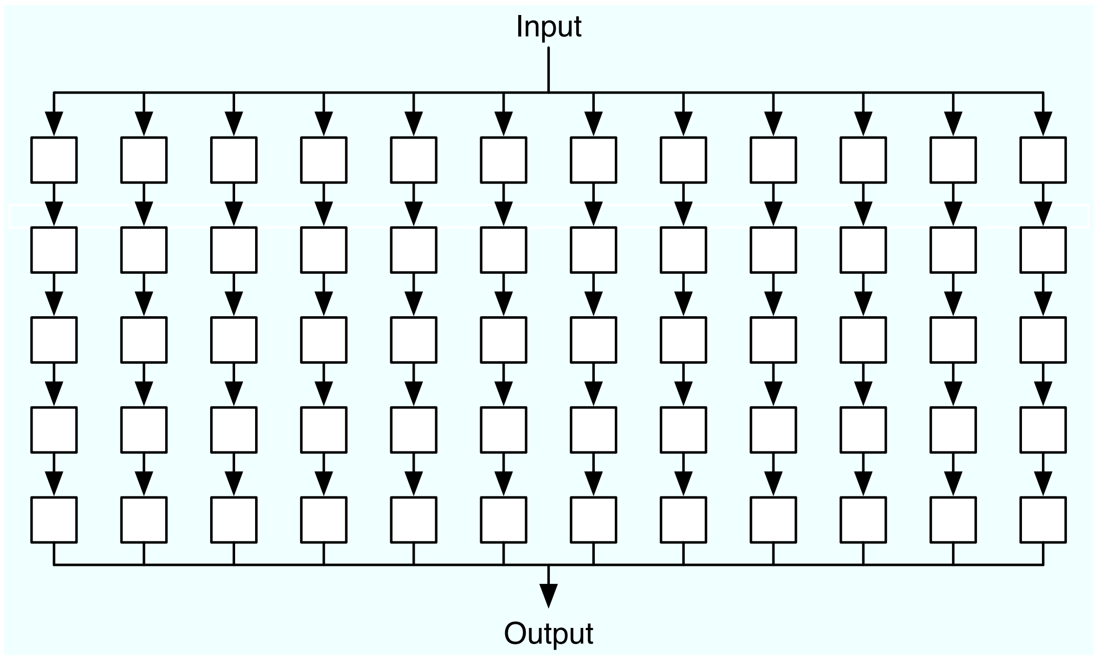


Figure 7.14: Pipeline parallelism

dimension, and is limited by the number of concurrent requests each individual machine is capable of handling.

Figure 7.15 shows a workflow which uses pipeline parallelism. In addition to the list size and the amount of computation per service call, its command-line arguments include a list of hosts on which to invoke services, each of which corresponds to a single stage in the pipeline. The input list is constructed in the same manner as in the data parallel workflow in Section 7.3.1. The top-level logic of the workflow, the expression bound to `result`, invokes the pipeline for every item in the input list. The `pipeline` function starts with a call to the last processing stage, passing in as input the result of the previous stage, which takes input from the stage before it, and so forth, until the initial input to the first stage comes from the value passed as the `arg` parameter. Because the arguments to `callservice` must be evaluated before the actual call takes place, the stages are carried out in order from first to last.

Since the total amount of computation involved with this workflow is proportional to the number of nodes, it does not make sense to measure its speedup in the same way as in the other experiments, which involve varying the number of nodes while the total amount of computation stays the same. Instead, we performed a series of runs with different list sizes, using orchestration, with a fixed number of eight nodes/stages. Each service call performed ten seconds of computation, and the initial input to the first processing stage was the empty string. The reason why we used the same amount of computation for each stage is that even when processing times differ, the total amount of time required for all items to pass through a pipeline is inherently limited by the longest stage, though machines executing shorter stages would have lower processor utilisation due to the time they spend waiting for the longer stages.

```
main args =                                // Main function
(if (< (len args) 4)                       // Check command-line args
  "Usage: pipeline.elc nitems comp port itemsize services...\n"
(letrec
  nitems = (ston (item 0 args))            // Process command-line args,
  compms = (ston (item 1 args))            // using ston to convert strings
  port = (ston (item 2 args))              // to numbers for numeric args
  itemsize = (ston (item 3 args))
  services = (skip 4 args)                 // Remaining args: service hosts

  input = (genstring itemsize)             // Generate input string of
                                           // requested size

  inputlist = (map (!n.input)              // Create input list with n
                   (range 1 nitems))       // copies of input string

  callservice host value =                 // Service stub function:
    (tail                                  // Ignore first response char (.)
    (connect host port                     // Connect to specified server
    (mkrequest compms value)))             // and send request

  pipeline stages arg =
    (if stages                             // If more stages remain, call
        (callservice                       // the service for the current
          (head stages)                    // stage, passing the result of
          (pipeline (tail stages) arg))    // the previous stage as input
        arg)                               // Else return initial input

  result = (map (pipeline services)        // Invoke pipeline for each item
                inputlist)                 // in input list
in
  (foldr ++ "END\n" result)))              // Print result list, terminated
                                           // by "END" to confirm completion

mkrequest compms value =                   // Produce request body: amount
(++ (ntos compms) (++ " " value))          // of computation + input value
```

Figure 7.15: Workflow for testing pipeline parallelism

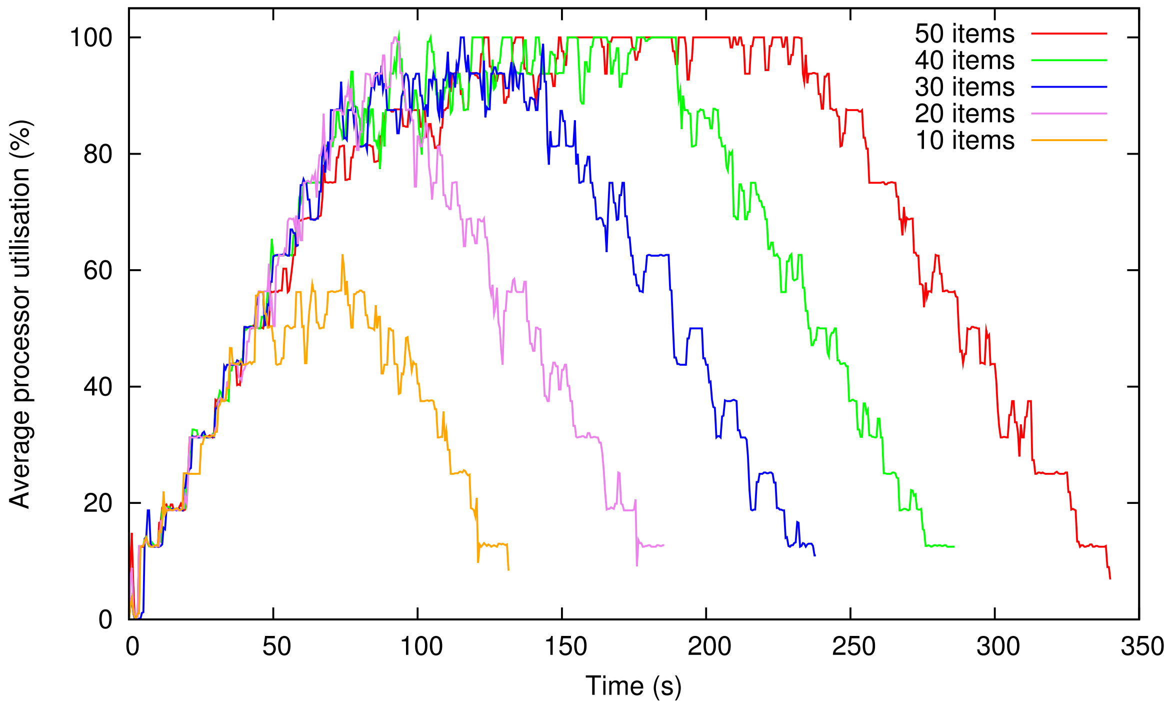


Figure 7.16: Average processor utilisation with physical pipelining (8 nodes, 8 stages)

Figure 7.16 shows the average processor utilisation of the nodes during execution. This graph illustrates an important an important performance limitation inherent in the nature of physical pipelining. Since each machine is only responsible for a single processing phase, most of the machines remain idle near the start of execution, since there are no items available yet which are ready to go through the corresponding processing stage. As time progresses, more items progress through the pipeline, and become ready to undergo the processing involved with later stages. Eventually, the pipeline fills up, and all machines become fully utilised. At the other end, the reverse happens — as the number of items remaining to be processed by each stage reaches zero, there is no work left for the machines earlier in the pipeline to do. This causes them to become idle, even though machines later in the pipeline are still kept busy with the last few items.

Efficient utilisation of processors in the context of physical pipeline parallelism is only possible when the number of items is significantly greater than the number of pipeline stages. When this is the case, the idle periods at the beginning and end of execution occupy only a small portion of the total execution time, and thus on average, close to full utilisation can be achieved over the lifetime of the workflow.

The processor utilisation shown in Figure 7.16 is essentially what one would expect to see for a correctly-functioning implementation of pipeline parallelism, demonstrating that NReduce successfully detects all of the available parallelism. At the beginning of execution, the call to `map` triggers evaluation of the `pipeline` function for every item in the input list. Each of these calls demands evaluation of the last stage in the pipeline, which in turn demands evaluation of the second-to-last stage, and so on, until the first stage is finally demanded. These demands are all propagated at the beginning of execution, causing sparks to be generated for the first stage of processing for all list items.

As soon as the first machine in the pipeline finishes processing a service request, the function call that was waiting for its completion is woken up, and then proceeds to add a spark to the pool corresponding to the following stage of the pipeline. This spark will be sent to the appropriate machine for the next stage in the pipeline as soon as it becomes idle, after finishing another request. Since the first machine is now idle, the spark for the next item in the list will be activated, causing the first stage of processing for that item to begin. The result of this behaviour is that list elements proceed through the processing stages in sequence, with other elements following as machines become available to process them. This continues until all of the stages have been completed for all items.

7.3.5 Logical pipelining

Although pipeline processing is common in workflows, its direct realisation in a physical configuration where each machine can only process one stage limits the extent to which all machines can be used to their full capacity. This is a problem when the number of items is small, or the number of stages are large. Physical pipelining is unavoidable in situations where a user is restricted to already-deployed services located on different machines, but if it is possible to have all services deployed on all machines, the problem can be avoided.

With logical pipelining, all machines can be used right from the beginning, because initially all of the items need to be handled by the first stage, and this can be done by any machine. Similarly, towards the end of execution, all machines can be involved with the execution of the final stages of processing. This eliminates the periods at the beginning and end of execution where some machines are idle. The behaviour of logical pipelining is similar to that of data parallelism, except that there are multiple, independently-schedulable operations performed for each item. Although the behaviour of this test is similar to that of the data parallel test described previously, we included it to show how performance of a workflow based on pipeline processing could be improved purely by changing the deployment of the services used so that they are available on all machines.

To demonstrate the superior utilisation achievable with logical pipelining, we re-ran the utilisation experiment from the previous section, except this time allowing any processing stage to occur on any machine. As in the previous section, the workflow shown in Figure 7.15 was run in orchestration mode with several different list sizes and a fixed count of eight pipeline stages on eight nodes, with ten seconds of computation per stage. In this case, we specified the command-line arguments such that each service call was directed to a load balancer, which distributed the requests to each of the machines. Figure 7.17 shows the average processor utilisation of the nodes during this experiment, which can be compared with Figure 7.16 from the previous section.

As can be seen from the graph, all machines are at 100% processor utilisation almost the entire time, with the exception of the 10-item test, which does not contain enough parallelism to keep all of the processors busy. In contrast with the physical pipeline test,

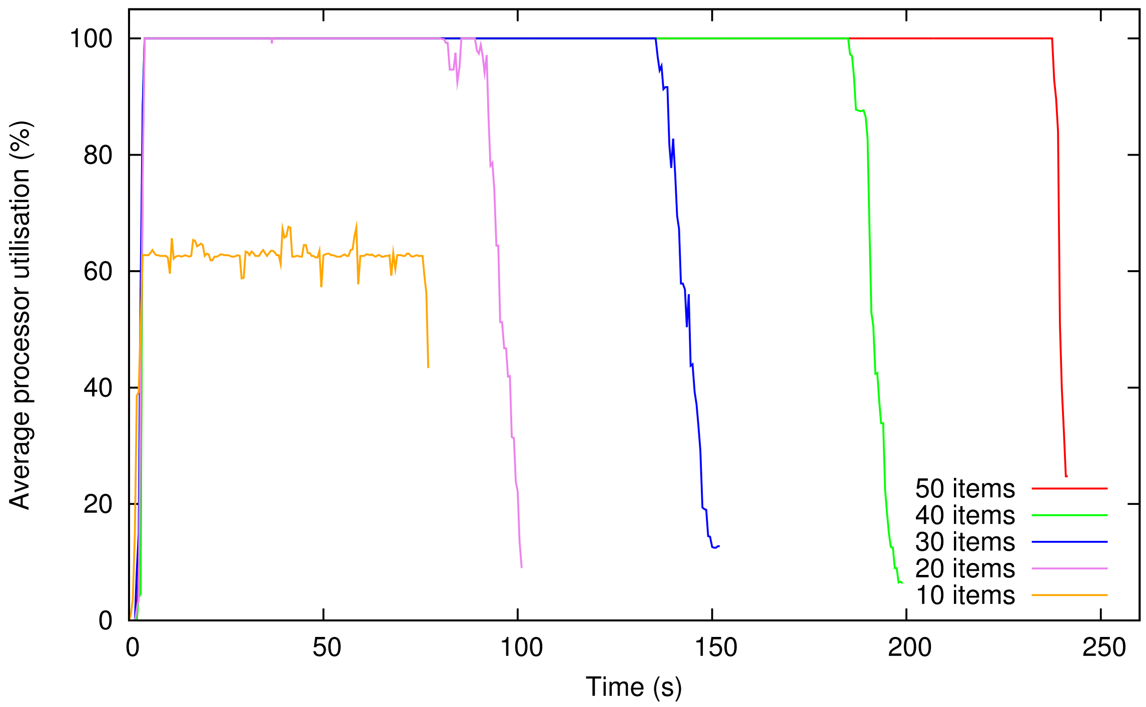


Figure 7.17: Average processor utilisation with logical pipelining (8 nodes, 8 stages)

the idle periods at the beginning and end of execution are not present, and in all cases the execution completes sooner. For example, a pipeline of 50 items completes in just under 250 seconds, whereas in the previous case it took nearly 350 seconds.

Since this version can use any number of nodes, we tested the scalability as the number of nodes increases. Figure 7.18 shows the speedup for two experiments involving the workflow, both of which used 256 items with 5 pipeline stages. The first experiment used one second of computation, and passed empty strings between the services. The second experiment used five seconds of computation, and 256 Kb items. As the figure shows, high levels of speedup are achieved in all cases, and the results are similar to those given for (nested) data parallelism in Sections 7.3.1 and 7.3.2. Because there were a smaller number of items here than for the data parallel tests, the data-intensive experiment achieved better speedups due to less time being required to collate the output data.

The way in which sparks are detected in this experiment is the same as that described in the previous section. The only difference is that here, each service call can go to any machine, so multiple sparks within a given processing stage can execute in a data parallel fashion. The mechanisms for detecting when a server at a particular IP address is able to accept more requests, as described in Section 6.2.4, effectively exploit the additional parallelism that comes from having all of the machines able to process any stage.

There is no penalty for using logical parallelism, since it only increases the extent to which the processors can be effectively utilised. For long lists involving stages of equal length, physical pipelining and logical pipelining can both achieve nearly full utilisation of all

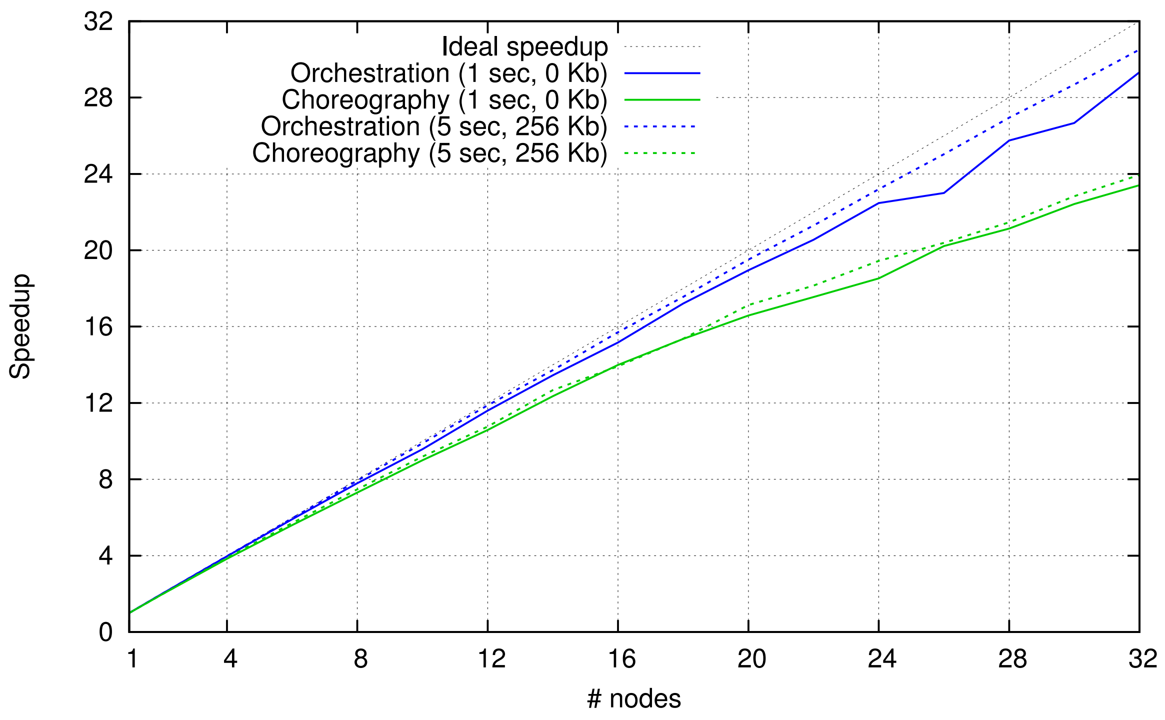


Figure 7.18: Logical pipeline parallelism — speedup (256 items, 5 stages)

processors. For shorter lists, or those involving different amounts of computation in each stage, it is better to use logical pipelining. In cases where the amount of computation in each stage differs, machines which would be under-utilised in physical pipelining can instead share the burden of executing the longer tasks. The dynamic scheduling mechanism makes this possible by assigning work to machines as it becomes available, rather than statically determining the assignment of tasks to machines.

7.3.6 Multi-stage data parallelism

The last type of parallelism we looked at is a variation on data parallelism in which a list of items goes through several stages, each of which consists of data parallel processing followed by a sequential collective operation involving the list as a whole. This differs from a pipeline structure in that the separate processing stages cannot run in parallel, due to the fact that the collective processing acts as a form of barrier synchronisation. Parallelism only exists *within* a stage, where multiple items can be processed at the same time. This structure is shown in Figure 7.19.

One of the reasons we chose to look at this type of algorithm, other than showing how multiple instances of the data parallel approach can be combined, is to carry out scalability measurements for programs which have a sequential component. During execution, programs of this form alternate between parallel and sequential sections. Each time a new parallel section is entered, there is a short ramp-up period during which all of the parallel

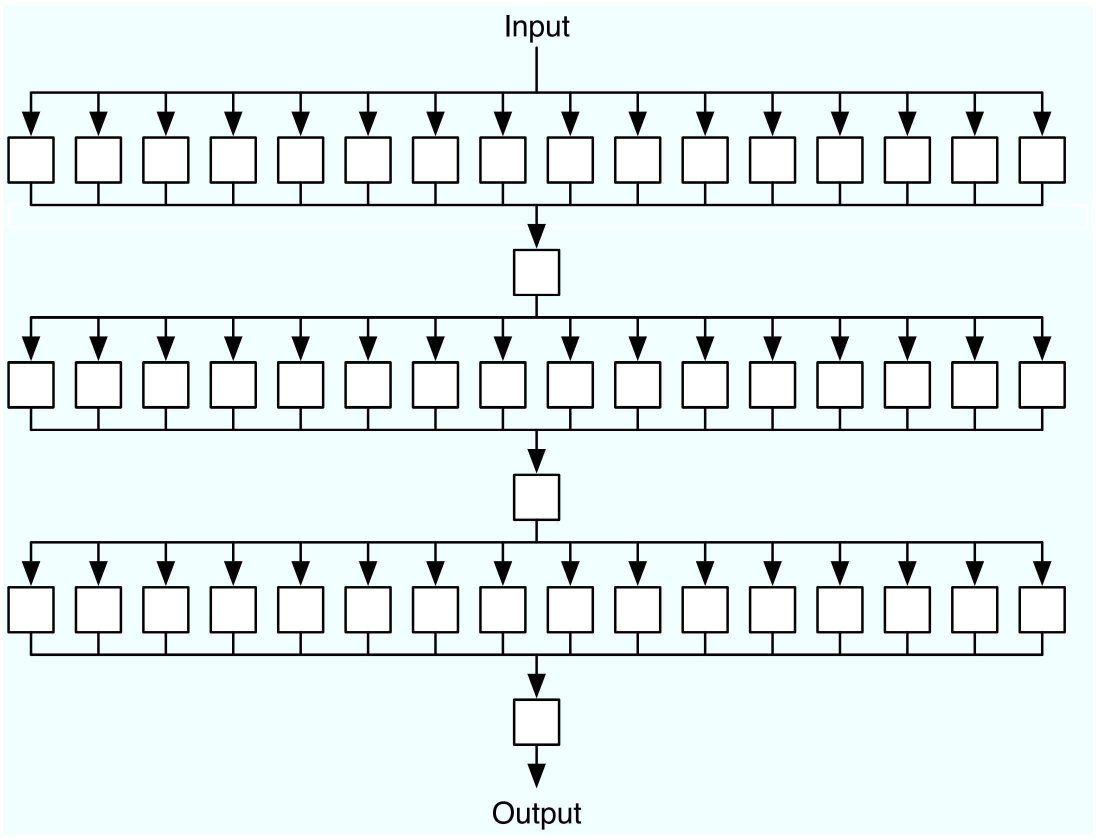


Figure 7.19: Multi-stage data parallelism

calls start, and a tear-down period in which the collective processing barrier forces the program to wait until all parallel calls have completed.

The performance of parallel programs with a sequential component can theoretically be predicted by Amdahl's law [12], which states that the speedup achievable for a program is given by:

$$\frac{1}{(1-P)+\frac{P}{N}}$$

where P is the proportion of computation that can be executed in parallel, and N is the number of processors. This model assumes an ideal situation where the transition between parallel and sequential sections is instant, or is at least insignificant compared to the total execution time. However, as the ramp-up and tear-down periods become more significant, the difference between the actual speedup and that predicted by Amdahl's law becomes greater. Comparing the predicted and actual speedup can thus give an insight into how quickly the system is able to make use of new parallelism once it becomes available.

Figure 7.20 shows the workflow that we used to test multi-stage data parallelism. Its operation is very similar to the data parallel program from Section 7.3.1. The difference with this version is that it performs the data parallel processing several times (based on

```
main args =                              // Main function
(if (< (len args) 5)                     // Check command-line args
  "Usage: msdp.elc host port nitems nstages compms itemsize\n"
(letrec
  host = (item 0 args)                   // Process command-line args,
  port = (ston (item 1 args))            // using ston to convert strings
  nitems = (ston (item 2 args))          // to numbers for numeric args
  nstages = (ston (item 3 args))
  compms = (ston (item 4 args))
  itemsize = (ston (item 5 args))

  input = (genstring itemsize)           // Generate input string of
                                         // requested size

  process data index =                   // Service call: single item
    (tail                                // Ignore first response char (.)
    (connect host port                   // Connect to service
    (++ "process "                       // Send "process" command
    (++ (ntos compms)                    // Send requested compute time
    (++ " " data)))))                    // Send input data

  combine results =                      // Service call: collective op
    (tail                                // Ignore first response char (.)
    (connect host port                   // Connect to service
    (++ "combine "                       // Send "combine" command
    (++ (ntos compms)                    // Send requested compute time
    (foldr ++ ""                         // Send concatenated results of
    (map (!x.++ " " x) results))))))     // all individual process calls

  loop value nstages =                   // Iterate over stages
    (if (> nstages 0)                    // If more stages remaining
        (loop (combine                   // Recursive call, with input
                (map (process value)     // from applying combine to all
                     (range 1 nitems)))  // results of process calls
              (- nstages 1))
        value)                           // Otherwise, return last result

  result = (loop input nstages)          // Start iteration
in
  (++ result "\nEND\n")))                // Print result, terminated by
                                         // "END" to confirm completion
```

Figure 7.20: Workflow for testing multi-stage data parallelism

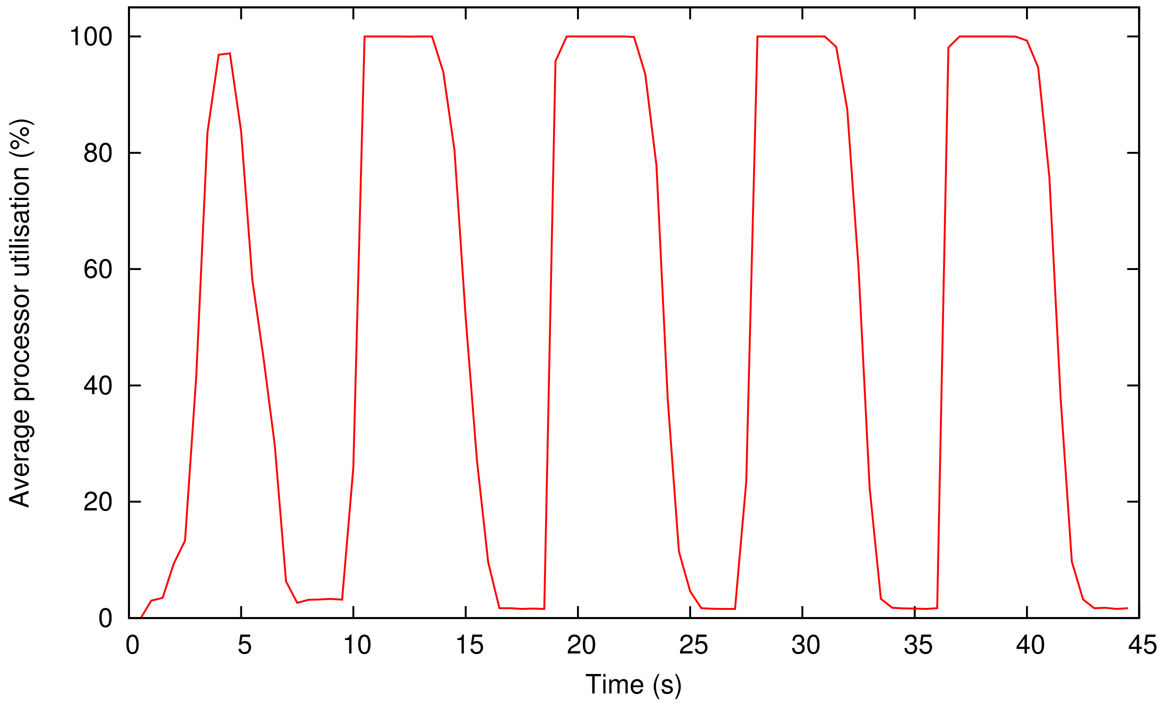


Figure 7.21: Average processor utilisation for multi-stage data parallelism (32 nodes)

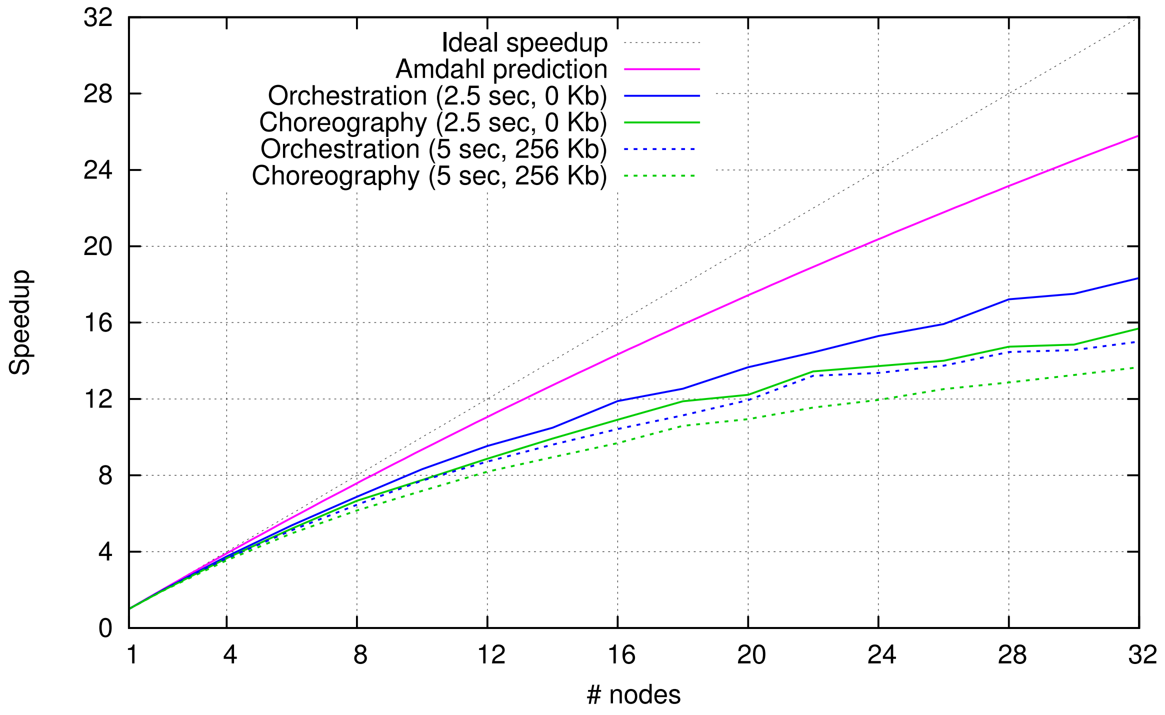


Figure 7.22: Multi-stage data parallelism — speedup (128 items, 5 stages)

the number of stages specified on the command line), and makes an additional service call between stages, which performs some compute-intensive aggregate processing on the results of all the previous operations.

We use a variation of the generic computation service described in Section 7.1, which accepts two types of requests. Individual processing requests begin with the string “process”, followed by the number of milliseconds of computation to perform, and then arbitrary data, which is echoed back by the service. Collective processing requests begin with the string “combine”, followed by the number of milliseconds of computation to perform, and a space-separated list of values, all of which must be of equal size. The amount of data sent back by the combine operation is equal to the size of an individual value. These details of service input and output are mostly of relevance when comparing data transfer between orchestration and choreography, which we examine for this workflow in Section 7.4.4.

To better understand the implications of the ramp-up and tear-down periods that happen during execution, consider the utilisation plot shown in Figure 7.21. This was obtained by running the program in orchestration mode across 32 dual-processor nodes, with a list size of 128 items, 5 processing stages, and 2.5 seconds per service call. The sequential/parallel sections can clearly be seen, where the average processor utilisation drops to 1/64th during the sequential sections, and goes up to 100% during the parallel sections.

As can be seen from the graph, the ramp-up happens almost instantaneously. Once the collective operation completes for a given stage, the frames for all items in the following stage immediately become sparked, and the virtual machine quickly establishes the maximum number of allowed connections to the servers. At the end of data parallel processing however, the tear-down period takes nearly three seconds. This is due to a slight load imbalance — all it takes is for one machine to get one more service call than the average, and the collective processing stage is delayed by 2.5 seconds while this completed, during which time all the other machines remain idle. It can be expected that the relative impact on overall execution time would be less if there were a larger number of requests, or shorter service calls.

Figure 7.22 shows the speedup obtained for two experiments carried out for this workflow, both of which used 128 items and 5 stages. The first experiment carried out 2.5 seconds of computation per service call (as in Figure 7.21), and used empty list items. The second experiment carried out 5 seconds of computation per service call, and used 256 Kb list items. Included in the figure is the speedup as predicted by Amdahl’s law, with a value of $P = 128/129 = 0.99249$, since for every 128 individual item calls, there is one collective call, which has the same computation time as the other calls. As the graph shows, the speedup obtained by both orchestration and choreography is less than that predicted, due to the influence of the ramp-up and tear-down periods at the start and end of each parallel section. The speedup for choreography is slightly lower than that of orchestration, due to scheduling overheads. Both orchestration and choreography were slower for the second test, due to the extra time required to transfer data across the network.

7.3.7 Summary

In this section, we have demonstrated the following:

- NReduce caters for many forms of parallelism that are in common use. It does this not by supporting individual types of parallelism explicitly, but instead by providing a generic mechanism for detecting and exploiting the parallelism available within a program. Algorithmic structures such as data parallelism, divide and conquer, and pipeline processing can easily be expressed in terms of lambda calculus, and automatically scheduled in such a manner as to effectively utilise multiple external services in parallel.

- The speedup levels achieved are close to what would be expected for each type of workflow structure, given the amount of parallelism available. In some cases, the speedup is inherently limited by the structure of the algorithm itself, such as the latter stages of divide and conquer processing, in which there is less parallelism available. In other cases, perfect speedup is theoretically achievable, and our system is often able to achieve near-perfect speedup on these, though the amount of data transfer can affect the speedup in some cases.

- In most cases, orchestration and choreography performed similarly. For the high-bandwidth network used in our experiment, choreography carries slightly higher overheads, due to its work distribution mechanism incurring delays due to work requests traveling between machines, whereas orchestration allows an external load balancer to make scheduling decisions immediately. However, by reducing the amount of data that is transferred between machines, which we study in the next section, choreography opens up the possibility of improving execution times in the context of slower network connections, particularly when the nodes are connected to each other via the Internet.

7.4 Data transfer

Choreography opens up the ability of improving the performance of workflows in low-bandwidth, wide-area networks, by reducing the amount of data that must be transferred across the network. Whereas previously, we studied the speedup our system can obtain for different types of parallelism, we focus here on measuring the extent to which choreography can reduce the amount of communication between machines, relative to orchestration.

In this section, we consider only the data-intensive experiment for each type of parallelism, for which choreography opens up the possibility of reducing the amount of data transferred between machines. All of the results reported here are taken from the same runs of the experiments discussed previously, with the exception of the physical pipelining experiment. We do not consider the flat or nested data parallel workflows here, since these do not involve any transfer of data between service operations. The other types of workflows all stand to benefit from choreography, and, as we demonstrate below, can see significant reductions in network communication compared to orchestration.

When we refer to network communication, what we mean is the total amount of data transferred between all of the machines, excluding any local communication that occurs between a virtual machine node and a service hosted on the same machine. Our measurements were taken by having each virtual machine node maintain a count of the total number of bytes sent to and received from other machines. The results were collated such that the exchange of data between each pair of machines was only counted once; the collation process verified that the sent and received byte counts recorded by both nodes in each pair were equal.

For all of the experiments discussed in the following sections, the expected outcome was that the amount of data transferred between machines should be significantly less for choreography than for orchestration. With the exception of physical pipelining, discussed in Section 7.4.2, the amount of network communication for orchestration should be independent of the number of nodes, since all requests and responses travel across the network between the client and the individual services. The amount of network communication for choreography may differ depending on the number of nodes, due to differences in distribution of frames between machines. As before, all results presented here are the averages taken over three runs.

7.4.1 Divide and conquer

Divide and conquer workflows stand to benefit from choreography, because they involve many cases where results are passed from one service call to another. With orchestration, this requires the results to be sent back to the client, whereas with choreography, the data can be sent directly to the node which takes it as input. Additionally, when two calls occur on the same machine, there is no need to send data across the network at all. Together, these factors can greatly reduce the total amount of network communication that occurs during execution.

Figure 7.23 plots the data transferred between machines against the number of nodes during the data-intensive experiment from Section 7.3.3, for both orchestration and choreography. This experiment uses an initial list size of 512 items, each of which is a string containing 64 Kb of data. There are nine stages of combining in total, which take as input 512, 256, 128, 64, 32, 16, 8, 4, and 2 items respectively. Since the result of the combine operation is the same size as its inputs, each stage consumes and produces 512 $\times$ 64 Kb = 32 Mb of data, so in total there is 9 $\times$ 32 Mb $\times$ 2 = 576 Mb of data exchanged with services.

Orchestration transfers exactly this amount of data over the network, regardless of the number of nodes. For choreography however, the amount of communication between machines is less. In some cases, the data is transferred over the local network interface on the same machine, and in others, the data is sent directly from the source machine to the destination machine. No data exchanged between services ever needs to pass through the client.

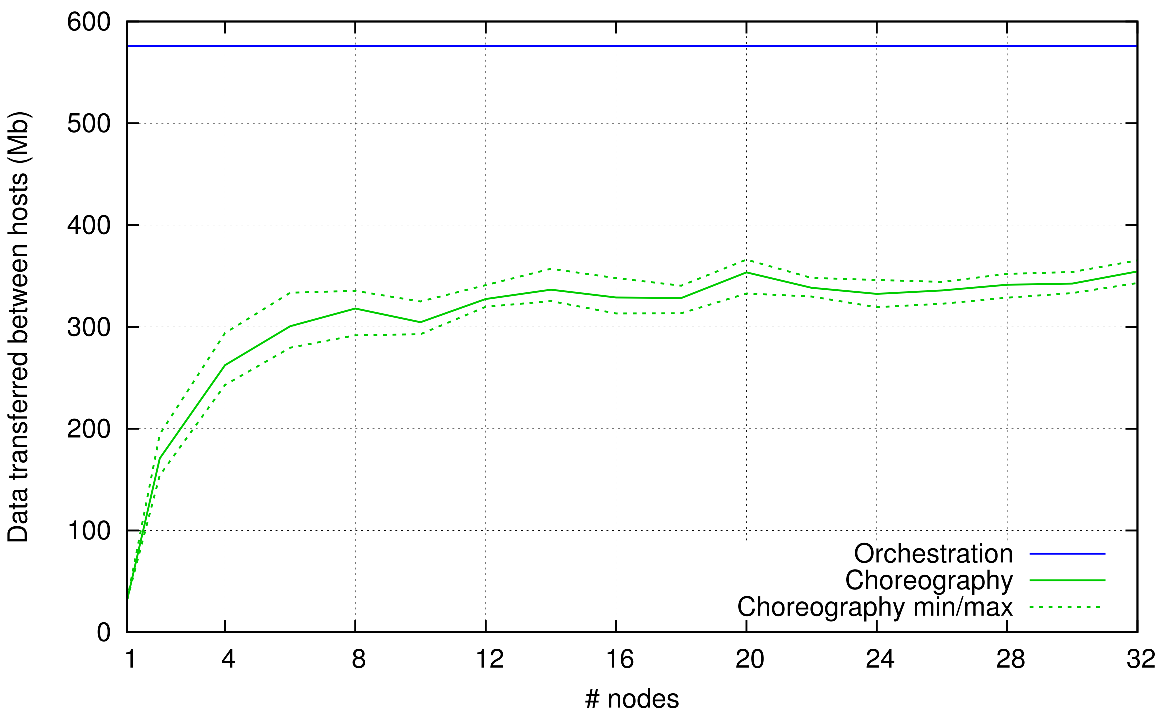


Figure 7.23: Divide and conquer — amount of data transferred across network

The dotted lines for choreography indicate the minimum and maximum values, with the solid lines indicating the average. The amount of network communication varied slightly for each of the three runs for a given number of nodes. This is because the assignment of specific frames to machines is non-deterministic, due to slight timing differences in the arrival of work request messages, as well as the fact that work requests move randomly from machine to machine until they find one or more sparks. Also contributing to the amount of network communication are the control messages used to manage graph distribution, scheduling, and distributed garbage collection, which in turn are influenced by which frames happen to be scheduled to which machines. These effects are also seen in the following sections.

The least amount of network communication occurs when only a single node is in use, in which case only 32 Mb is transferred, corresponding to the output of the workflow, which is sent back to the client. The reason this result is so low is that the service and workflow engine both reside on the same node, so all of the service calls occur over the local network interface. This involves just as much data transfer as for orchestration, but does not count towards the total shown in the graph, since the transfers are local.

Of course, using only a single node for choreography is pointless, because the same effect could be achieved using orchestration with the client residing on the same node as the service; also, large-scale parallelism is only possible when using many nodes. As the number of nodes increases, the total amount of network communication increases at first, but quickly stabilises at around 350 Mb. When only a few nodes are in use, the probability of a given pair of frames being assigned to different machines is low. If two frames are on

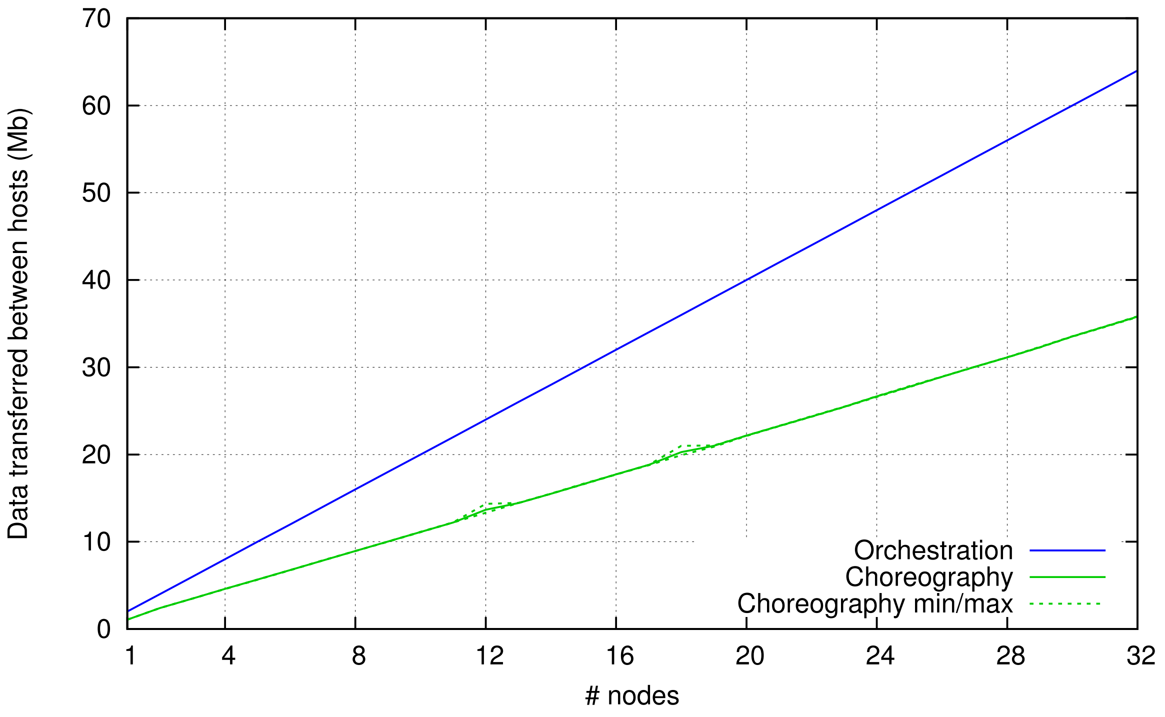


Figure 7.24: Physical pipelining — amount of data transferred across network

the same machine, and one uses the result of the other, the result is transferred locally. As the number of nodes increases, it becomes more likely that the frames will be placed on different machines, meaning the data must be transferred via the network.

7.4.2 Physical pipelining

In the physical pipelining experiment, each pipeline stage runs on a separate node, and thus the number of service calls is proportional to the number of nodes used. Running this test with different numbers of nodes thus varies the total amount of data exchanged between services. Figure 7.24 shows the results for a pipeline containing a single item, with varying numbers of stages/nodes. The initial item and result of each service call is one megabyte in size. Since there is only a single item, execution is sequential — in this particular case, we are interested solely in reducing bandwidth usage, rather than exploiting parallelism.

With orchestration, the total amount of data transferred is the amount of data passed between services, which is exactly twice the input size multiplied by the number of nodes. This is because each service call involves sending a one megabyte request and receiving a one megabyte response. With choreography, the data transfer is just over half of this value, since there are $n-1$ service-to-service data transfers for n stages, plus one additional transfer to send the result back to the client.

This graph shows an average of around 36.7 Mb of data transfer for the test with 32

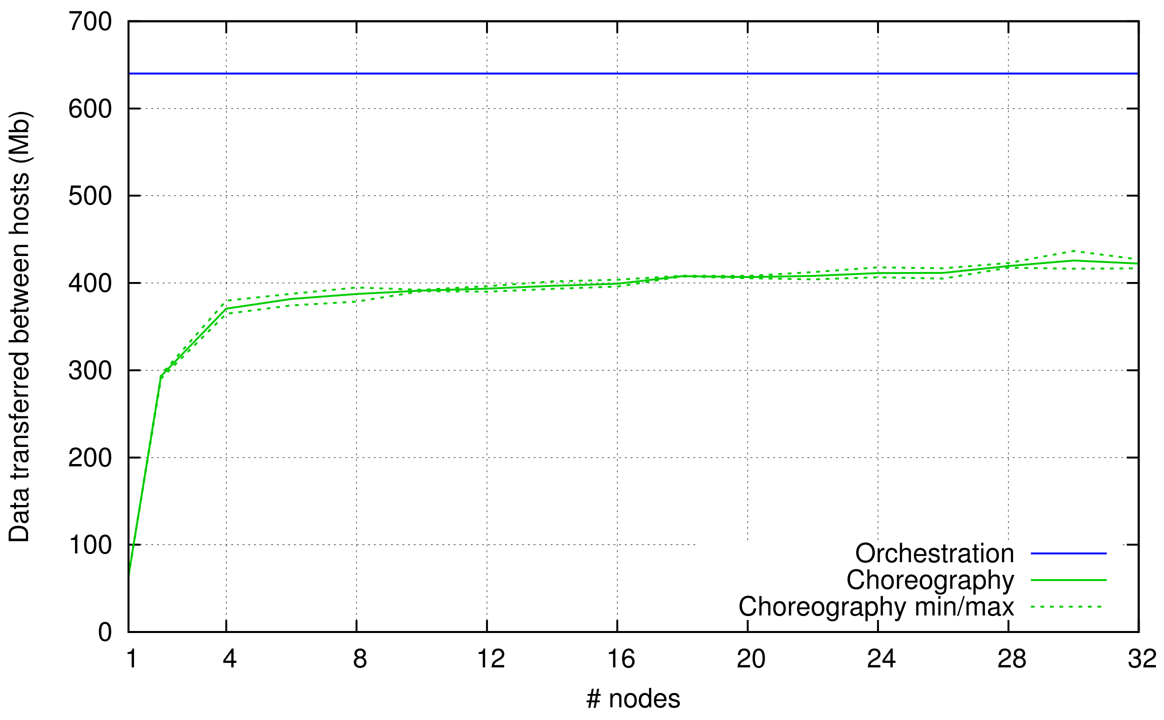


Figure 7.25: Logical pipelining — amount of data transferred across network

nodes. Intuitively, it would seem that only 32 Mb of transfer would be required, since there are 32 stages. The additional data transfer can be accounted for by two factors, both relating to overheads imposed by choreography. The first is that the compiled bytecode for the workflow must be sent from the client to each node. In this particular case, the bytecode is 51 Kb, so a total of 32 $\times$ 51 = 1,632 Kb must be sent in order for the execution nodes to have the code available. The second factor is the message passing that occurs between machines for work distribution, access to remote parts of the graph, and distributed garbage collection. There are 32 $\times$ 32 = 1,024 possible pairings of nodes. With an overhead of 36.7 - 32 = 4.7 Mb, minus the 1,632 Kb for bytecode distribution, this corresponds to around 3 Kb worth of control messages between each pair of nodes.

7.4.3 Logical pipelining

As discussed in Section 7.3.5, the logical pipelining approach permits any node to execute any processing stage. With choreography, this has the benefit that if two consecutive stages of processing for a given item are scheduled on the same machine, there is no bandwidth cost associated with passing the results between them. Such cases instead utilise the local network interface, which simply involves a memory-memory transfer. We expected that this would happen for all of the transfers between different processing stages for the same item, so that network communication would only occur for the input to the first stage and the output from the last stage. The results, however, showed otherwise.

For this experiment, we used a pipeline with five stages and a list size of 256, with each item being 256 Kb in size. Figure 7.25 shows the results for this workflow with varying numbers of nodes. Orchestration involves the transfer of exactly 640 Mb of data across the network, regardless of the number of nodes. This figure corresponds to 5 stages $\times$ 50 items $\times$ 256 Kb $\times$ 2 transfers (request and response).

For choreography, the amount of network communication for a single node is 256 items $\times$ 256 Kb = 64 Mb, since the initial list input is generated on the virtual machine node, and the resulting list items are sent back to the client, which runs on a separate machine. As the number of nodes increases, larger amounts of data must be transferred between nodes, since different pipeline stages are scheduled to different machines. After the first few nodes, the workflow quickly reaches a semi-stable state where the amount of data transferred between machines ranges from around 390 to 420 Mb. This figure was much higher than expected; we concluded that the results can be explained as follows:

The outermost function call for each item in the list corresponds to the last stage in the pipeline. Sparks are generated for all of these function calls, and when each of them executes, it will call the previous stage, which will call the stage before that, and so forth, until reaching the first stage in the pipeline. The first few calls to this will begin executing immediately on the initial node, and the others will be postponed, as discussed in Section 6.4. These postponed sparks will quickly be distributed among the other machines, as the initial work requests arrive. At this point, all of the machines will be busy executing the first pipeline stage. However, the calling frames for all of these remain on the first node.

Once a frame corresponding to a service call completes, a message is sent back to the calling frame on the initial node, indicating that the result is now available. This causes the caller to become runnable, and attempt to connect to the service on its local node. Often, there will already be the maximum number of calls in progress, so the frame will be postponed and subsequently scheduled to another machine. When the frame arrives at its destination, it will connect to the service on that node and send the input data, which must first be transferred from the machine on which the previous pipeline stage was executed. The net effect of this is that in most cases, each stage of processing for a given item ends up executing on a different machine to the previous stage, and this requires its input data to be transferred over the network.

This outcome is not ideal — it would be better to execute all processing stages for a given item on the one machine. However, our scheduling mechanism does not take data locality into account when deciding whether to migrate a frame. Thus, frames often end up moving to other machines, taking their input data with them, which requires more network communication than necessary. Nonetheless, the results obtained here are close to those obtained for physical pipelining in the previous section, due to data being transferred directly between service hosts instead of via the client.

One area of future work regarding scheduling is to make it locality-aware. This would require the location of data referenced by a given frame to be considered when deciding which machine to assign the frame to. With our current scheme, this would be difficult, because at present, decisions about which frames to migrate are made in response to work requests from individual machines. Consideration of locality would be easier if the

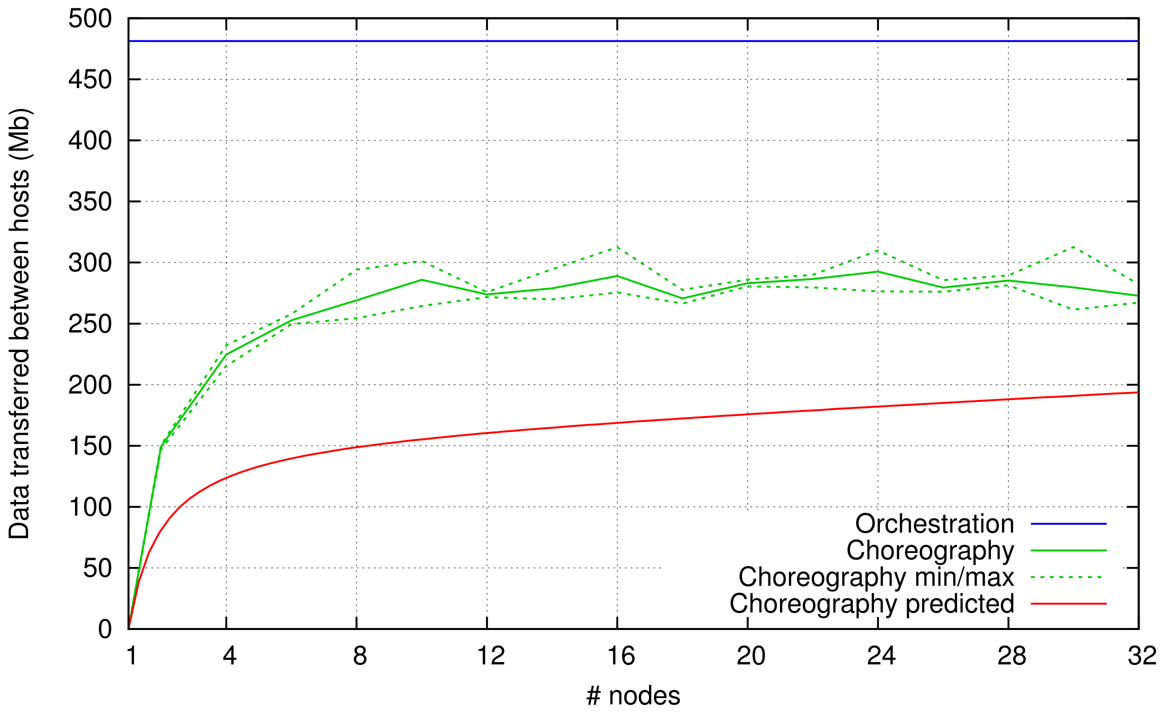


Figure 7.26: Multi-stage data parallelism — amount of data transferred across network

scheduling mechanism allowed each node to be aware of all other machines that are idle. If this were the case, then for each spark, it could decide which idle node would be the best to assign the spark to.

For workflows involving pipeline processing, where the user has control over the deployment of services, it would be preferable to merge all of the stages into a single operation, and then invoke them using a data parallel workflow. This would ensure that all exchange of data between different pipeline stages for a given item occurs on the local machine. However, this is not an option when the user is composing existing services that are already deployed, such as third-party services at a remote site over which the user has no authority. There is no way for the virtual machine to force multiple service operations to be merged together into a single operation, since this is an issue relating to the way in which the services are implemented.

7.4.4 Multi-stage data parallelism

Finally, we looked at the data transfer reductions achievable for multi-stage data parallelism. Section 7.3.6 discussed the multi-stage data parallel workflow we used for our experiments, and gave speedup results for this workflow. Here, we examine the network transfer involved with the data-intensive experiment discussed in that section. This experiment uses a list size of 128 items, with 256 Kb per item, five stages of processing, and five seconds of computation per service call. Figure 7.26 shows the amount of data transferred over the network for both orchestration and choreography.

Recall that this workflow involves two different service operations. The `process` operation performs individual processing on a single list item, consuming and producing the same amount of data. The `combine` operation performs collective processing on the entire list, producing output whose size is equal to that of a single item. Thus, each stage involves 128×256 Kb $\times 2 = 64$ Mb of data for the `process` calls, and $128 \times 256 + 256$ Kb $=$ 32.25 Mb for the `combine` call. Over five stages, the workflow therefore exchanges a total of 481.25 Mb of data with services. As Figure 7.26 shows, this is exactly the amount of data transferred over the network for orchestration.

For choreography, we can predict the amount of network communication as follows: Assuming an even distribution of service calls among n nodes, each node would execute $128/n$ service calls during each stage. The `combine` operation must run on a single node, so the results of all service calls on other machines must be transferred to this node, totaling $256 \times (128 - 128/n)$ Kb of data. Since the output of the `combine` operation is used by all individual `process` operations in the following stage, and these operations will be distributed among all of the other nodes, this will involve a transfer of $256 \times (n - 1)$ Kb of data. The total amount of network transfer we should expect to see for choreography can thus be determined by the following formula:

$$\text{transfer}\,(n) \quad = \quad 5 \times (256 \times (128 - 128/n) + 256 \times (n-1))$$

For 32 nodes, this would be 193.75 Mb. Note that the values predicted by this formula exclude control messages, so we should expect to see slightly more than this amount transferred across the network.

Figure 7.26 shows the results for this experiment, including the predicted values. As can be seen, the actual results are significantly higher than the prediction. While this difference can be partially accounted for by control messages, we found that much of it was due to duplication of data that was supposed to be shared, as discussed in Section 4.6.3. It turned out that the array expansion logic designed to coalesce lots of small arrays into fewer big arrays was creating extra copies of the data in some cases. Disabling this mechanism caused performance problems on other experiments, due to a much larger number of messages being required to retrieve data split over lots of small arrays. The array expansion logic needs further tuning, a task we have deferred for future work. Nonetheless, choreography still achieves significant savings for this workflow relative to orchestration.

7.4.5 Discussion

All of the results obtained in this section showed substantial reductions in network communication for choreography, relative to orchestration. Although the exact ratio of network communication for orchestration and choreography differs somewhat between the different types of workflows, the savings offered by choreography are significant in all the cases we have tested. Each case showed a high level of scalability, in terms of achieving only gradual increases in data transfer with respect to the number of nodes. The results we have achieved suggest that our technique for implementing choreography is applicable to a wide range of workflows, enabling the use of a large number of nodes without incurring major overheads.

Since all of our testing has been done on a cluster, in which very high bandwidth is available, the reduction in data transfer does not significantly affect the execution times for the workflows in the above experiments. It is for this reason that we have focused on the volume of data transfer rather than completion time in these experiments. However, in Section 7.5.4, we demonstrate a scenario where even with the cluster's 100 Megabit network, it is still possible to improve execution times using choreography. For workflows utilising services at multiple sites connected via slower links, it can be expected that the reduced amounts of communication involved with choreography relative to orchestration would improve performance in the majority of cases. One aspect we have not addressed here is the impact of increased latency in such environments on the distributed execution logic — this is an area for future work.

7.5 XQuery-based workflows

To measure the performance of workflows expressed using XQuery, we developed an example case study involving an automatic marking system of the sort that might be used in a typical undergraduate computer science subject. The code for this workflow is shown in Figure 7.27. For a given assignment, students submit the source code for their solutions to a repository that is accessible to the marker via a web service interface. Once the due date for the assignment arrives, the marker runs the workflow, which performs an automatic marking process on each of the submissions.

The marking process works by going through all of the students, and for each, checking out and compiling the source code, then running the student's program with a series of test cases. Each test case consists of input data and output data. The workflow runs each test by invoking a testing service with the compiled code as well as the input and output data for that test. The service returns a true or false value indicating whether the output of the student's program matched the expected output for the test. Based on whether the test passed or failed, a certain number of marks is awarded. This is repeated for each test for every student.

Parallel execution is possible within this workflow due to the use of `for` loops on lines 8 and 12. The outer loop iterates over all of the students, and for each, invokes the `getSource` and `compile` service operations, all of which can run in parallel, although these are not computationally expensive and thus do not stand to gain much from parallelism. The inner loop invokes the `runTest` service operation for every test on every student's submission. All of these operations can also run in parallel, which is very useful in this case, since the tests are all computationally expensive.

```
import service namespace svc = "http://test.server/marks?WSDL";

<results>{
    let $students := svc:getStudents(),
        $tests := svc:getTests(),
        $maxmarks := sum($tests/marks)
    return
        for $s in $students
        let $source := svc:getSource(string($s/id)),
            $code := svc:compile($source),
            $testres :=
                for $t in $tests
                let $res := svc:runTest($code,
                                        string($t/input),
                                        string($t/expected))
                return <test marks="{if ($res = 'true')
                                     then $t/marks
                                     else 0}"/>,
            $total := sum($testres/@marks)
        return
            <student id="{$s/id}" name="{$s/name}">
                {$testres}
                <total>{$total}</total>
                <percent>{100.0 * $total div $maxmarks}</percent>
            </student>
}</results>
```

Figure 7.27: Student marks workflow

The result of the workflow consists of an XML document reporting the test results for each student, as well as their total mark and percentage. This document is constructed using the built-in XML element construction feature of XQuery, with expressions used where appropriate to calculate values such as the total marks. These calculations depend on the results obtained from the test execution service, indicating which tests a student passed or failed. Sample output for this workflow is shown in Figure 7.28.

This example is designed to demonstrate the following features of our system:

- Invocation of multiple service calls in parallel
- Minimisation of data transfer between machines, by avoiding re-transmission of compiled code where possible
- Internal computation and data manipulation within a workflow

```
<results>
   <student id="0" name="Student 0">
      <test marks="1"/>
      <test marks="2"/>
      <test marks="3"/>
      <total>6</total>
      <percent>100</percent>
   </student>
   <student id="1" name="Student 1">
      <test marks="1"/>
      <test marks="2"/>
      <test marks="0"/>
      <total>3</total>
      <percent>50</percent>
   </student>
</results>
```

Figure 7.28: Example output from the student marks workflow

- Production of XML result data based on results obtained from services

7.5.1 Speedup and data transfer vs. number of nodes

We measured the performance of this workflow by configuring the service with 64 students, 64 tests, and with the source code and compiled binaries each being 128 Kb for every student. The input and output sizes for the test cases were set to ten bytes each, and each test performed one second of computation. As before, we measured both orchestration and choreography; in this case were are interested in both the execution time and data transfer. The speedup obtained is shown in Figure 7.29, and the data transfer is shown in Figure 7.30; these results are the averages over three runs. The difference in speedup between orchestration and choreography was similar to that observed for the other tests, due to the additional scheduling overheads involved with choreography, as discussed in Section 7.3.1.

Regarding network communication, we can calculate the amount we should expect to see as follows:

- The `getSource` operation is called 64 times, returning 128 Kb of data each time, totaling 64 $\times$ 128 Kb = 8 Mb.
- The `compile` operation is called 64 times, each time taking 128 Kb of data as input, and returning 170 Kb of output (the compiled binary code is Base64 encoded, adding a 33% overhead). So for the `compile` call, the total amount of data exchanged with the service is 64 $\times$ (128 Kb + 170 Kb) = 18.6 Mb.

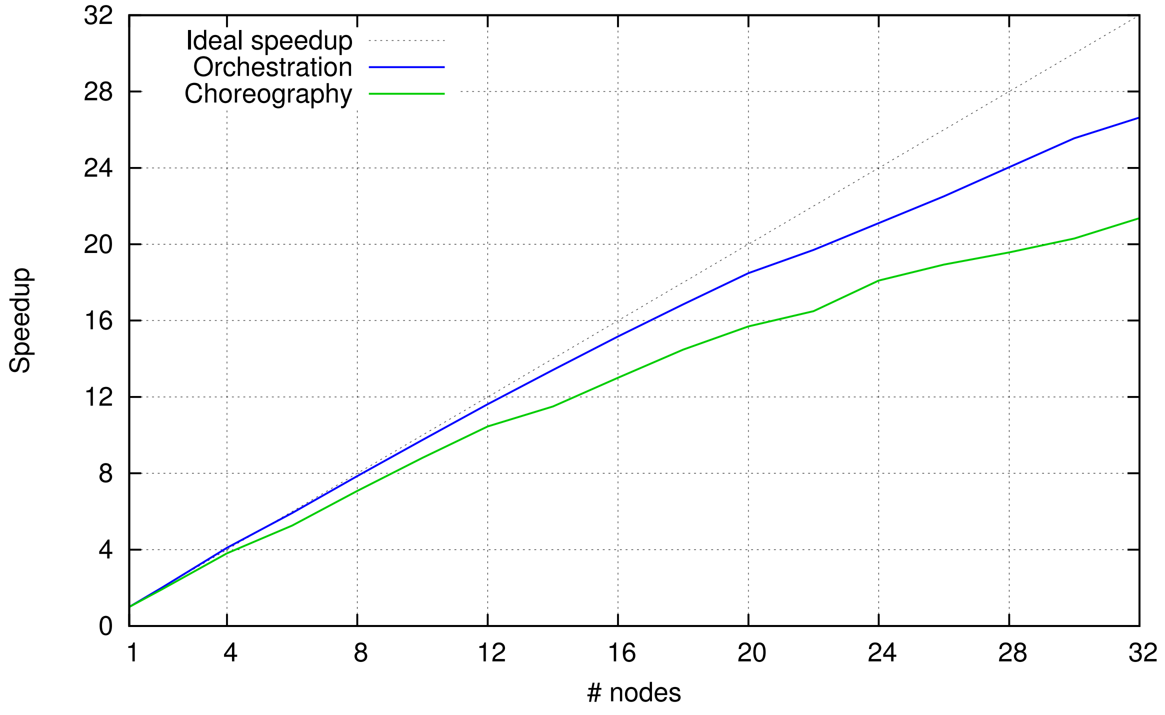


Figure 7.29: Student marks workflow — speedup

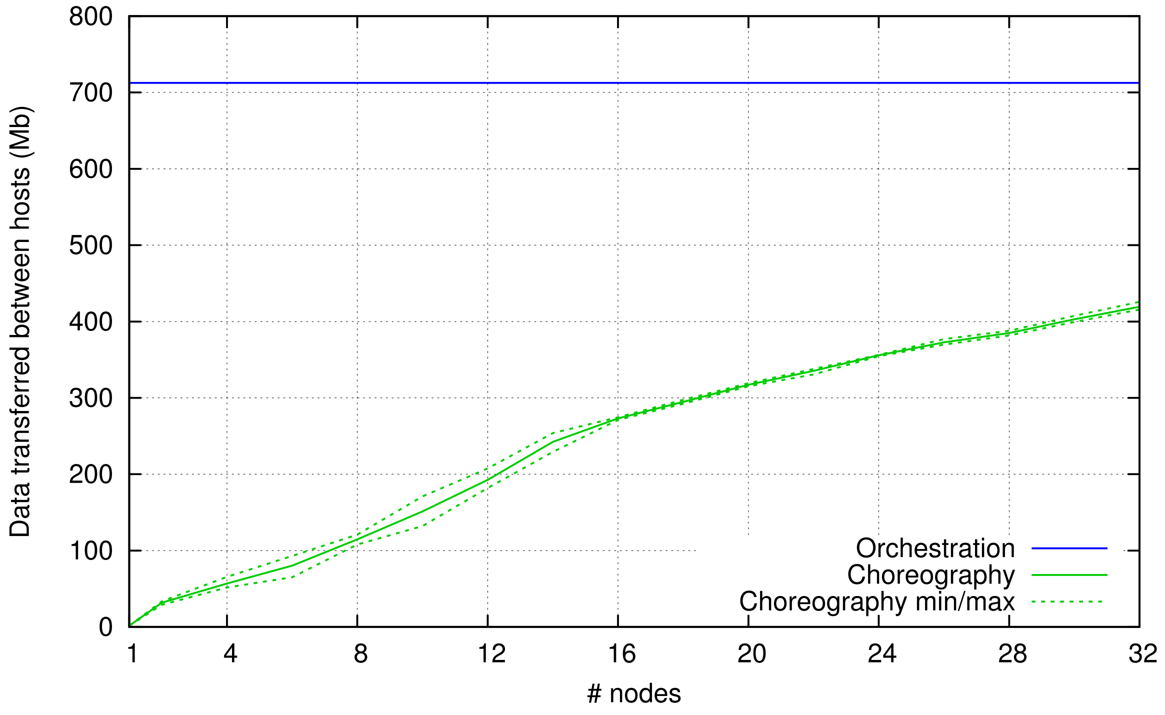


Figure 7.30: Student marks workflow — amount of data transferred across network

- The `runTest` operation, which runs the different test cases, is called 64 times per student, taking 170 Kb as input. For orchestration, the compiled code must be transferred over the network for every test, totaling $64 \times 64 \times 170$ Kb $= 680$ Mb. For choreography, the compiled code only needs to be transferred once for each machine it is used on, because all of the test cases that are run for a given student's submission share the same compiled code. In the worst case, every piece of code will be sent to every machine, so for 64 students and 32 machines, `runTest` would involve a maximum of $64 \times 32 \times 170$ Kb $= 340$ Mb of data transfer.

Based on the above, we should expect to see a total of $8 + 18.6 + 680 = 706.6$ Mb of data transferred across the network during orchestration, plus a small amount extra for HTTP headers and SOAP envelopes. The results in Figure 7.30 indicate that this is indeed the case. For choreography, we should expect to see up to $8 + 18.6 + 340 = 366$ Mb, plus the same overheads as for orchestration, as well as the control messages exchanged between nodes. The results shown in the figure are in line with this expectation.

The graph also shows that for choreography, the increase in the amount of network communication is approximately linear with respect to the number of nodes. The reason for this is the data transfer mentioned in the third point above — with high probability, at least one test case associated with a given student's submission will run on every machine, so the majority of compiled code objects need to be transferred to all machines. Since the number of compiled code objects remains constant, the amount of data that must be transferred across the network increases proportional to the number of nodes.

7.5.2 Speedup vs. granularity

In Figure 7.29, for 32 nodes, we see a speedup of around 26 for orchestration, and 21 for choreography. These results are reasonable, but not brilliant. However, it is important to keep in mind that this workflow uses relatively fine-grained tasks, each of which involves only one second of computation. To determine the speedup that could be obtained for coarser grained tasks, we ran another experiment which measured the speedup for 32 nodes with task sizes ranging up to eight seconds per `runTest` call. We also measured the speedup for finer-grained calls, of as little as 250 ms. The results are shown in Figure 7.31.

In each case, the total number of tasks was 4,096; the speedup was computed relative to a theoretical minimum time for one node of $4{,}096 \times n \ / \ 32 \ / \ 2$, where n is the number of seconds worth of computation per test. In practice, the time required to execute the workflow on a single node can be higher than this, due to overheads of coordinating the workflow and transferring the input and output data. The speedup results given here are therefore conservative, and slightly lower than in Figure 7.29. As can be seen, choreography achieves a speedup of 27.5 for tasks of four seconds, and a speedup of 29 for seven seconds and above.

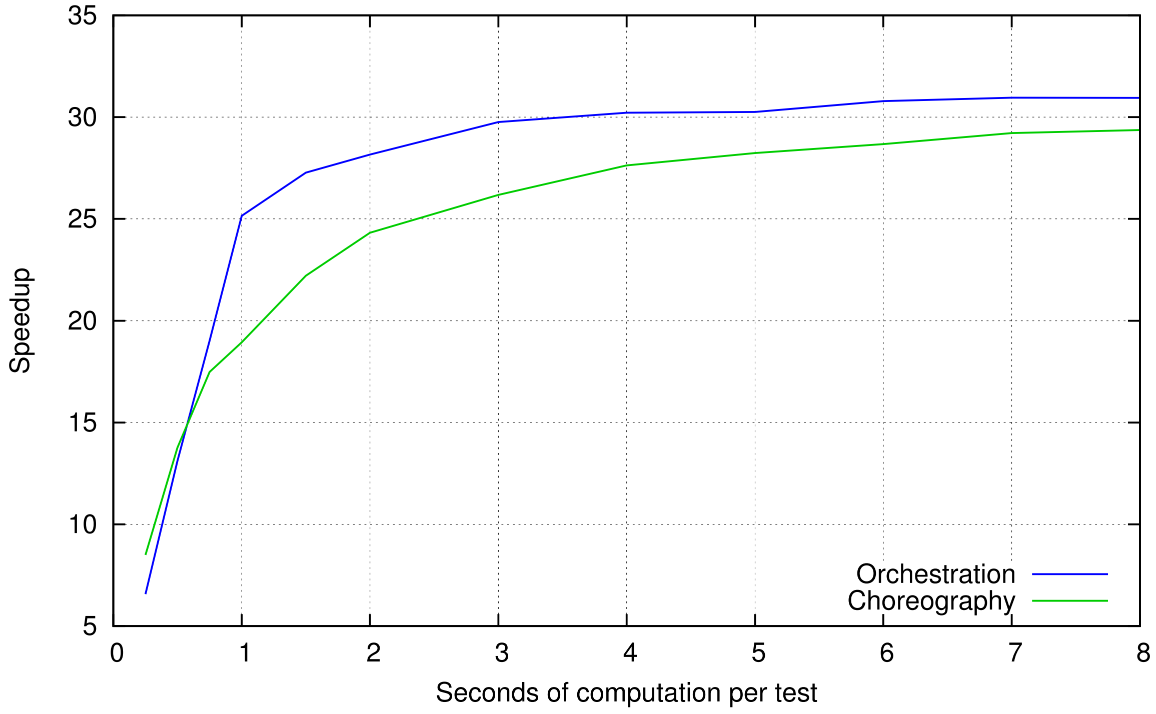


Figure 7.31: Student marks workflow — speedup vs. call granularity (32 nodes)

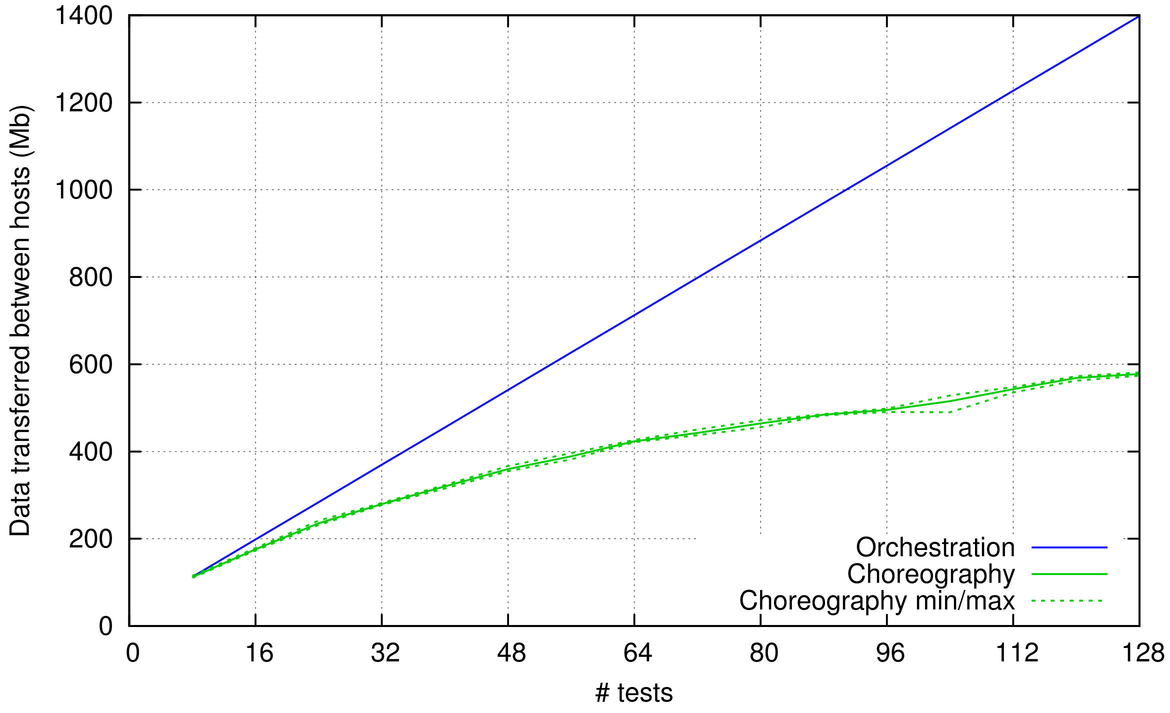


Figure 7.32: Student marks workflow — data transfer vs. number of tests (32 nodes)

7.5.3 Data transfer vs. number of tests

For this workflow, the reason why choreography results in less network communication than orchestration is as follows: Each piece of compiled code is often used multiple times on each machine. With orchestration, the code needs to be sent across the network every time it is used, but for choreography, it only needs to be transferred at most once to any given machine.

It follows from this that if the number of tests were to be increased, we should expect to see the amount of communication remain about the same for choreography, but increase linearly for orchestration. To test this hypothesis, we ran another experiment, in which we kept the the number of nodes fixed, while varying the number of tests. As before, this experiment used 64 student submissions, and one second of computation per `runTest` call.

Figure 7.32 shows the results of this experiment. With 64 tests per student, the results are the same as in Figure 7.30. Looking at the slope of the lines, we can see that the amount of communication increases linearly for orchestration, but much more gradually for choreography. Comparing the results for 64 and 128 tests, orchestration goes from 712 Mb to 1398 Mb — an increase of 96%. This is slightly less than double, since a small portion of the data is for the initial calls to `getSource` and `compile`, which execute the same number of times, regardless of how many tests are subsequently run.

For choreography, we see the results over this same range go from 423 Mb to 578 Mb — an increase of 36%. This increase is higher than we expected, a fact which can be attributed to the following factors: First, twice as many service calls are being made, so there are twice as many frames distributed among machines, resulting in more data being transmitted in work distribution messages. Second, since the workflow runs for a longer period of time, there are more work requests sent out by idle nodes. Third, the SOAP envelopes for each request and response are unique, and sometimes have to be transferred between machines, so the increased number of calls resulted in more of these being passed around. Nonetheless, the increase in communication for choreography over the 64 – 128 range is a great deal less than for orchestration, with choreography saving over 800 Mb of data transfer.

For smaller numbers of tests, the probability that a given node will receive the compiled code for a given student decreases, particularly when the number of tests per student becomes equal to or less than the number of nodes. The distribution of `runTest` calls which take a given code object is not necessarily even — some machines may receive many calls for a particular student's code, while others receive none. This contributes to the low rates of communication for very small numbers of tests in the case of choreography.

7.5.4 Impact of code size

Another aspect of interest is how the amount of data transferred between machines varies according to the code size. We should expect this to be linear — for orchestration, because

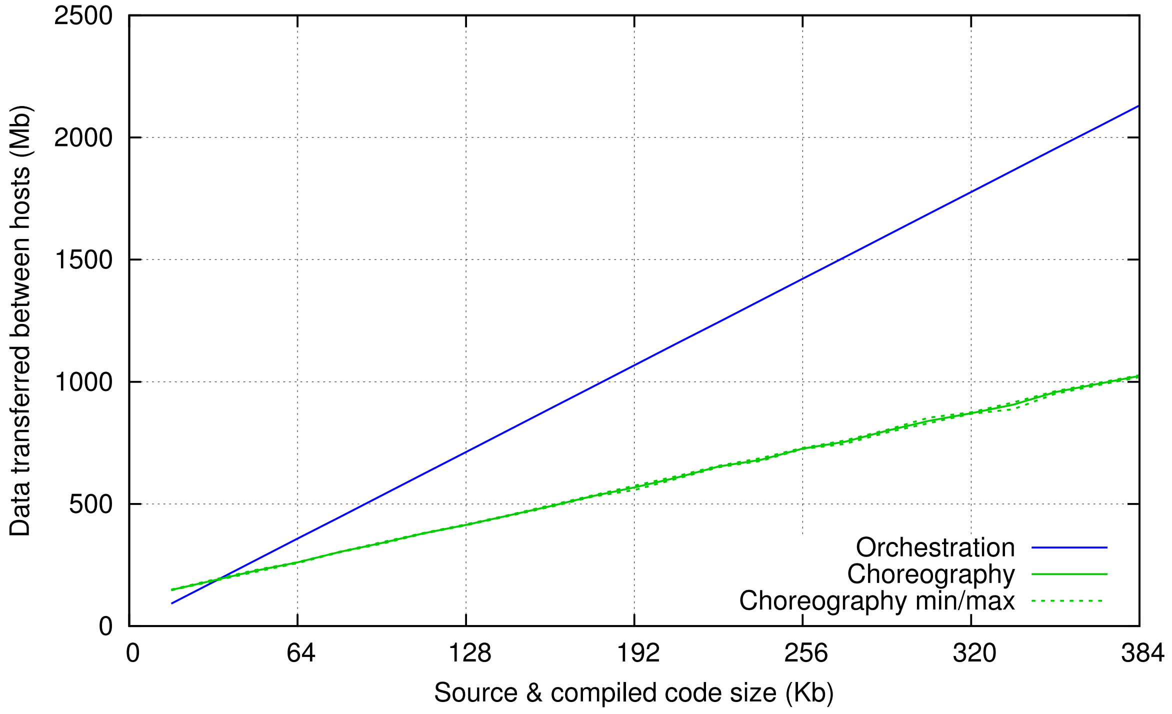


Figure 7.33: Student marks workflow — data transfer vs. code size (32 nodes)

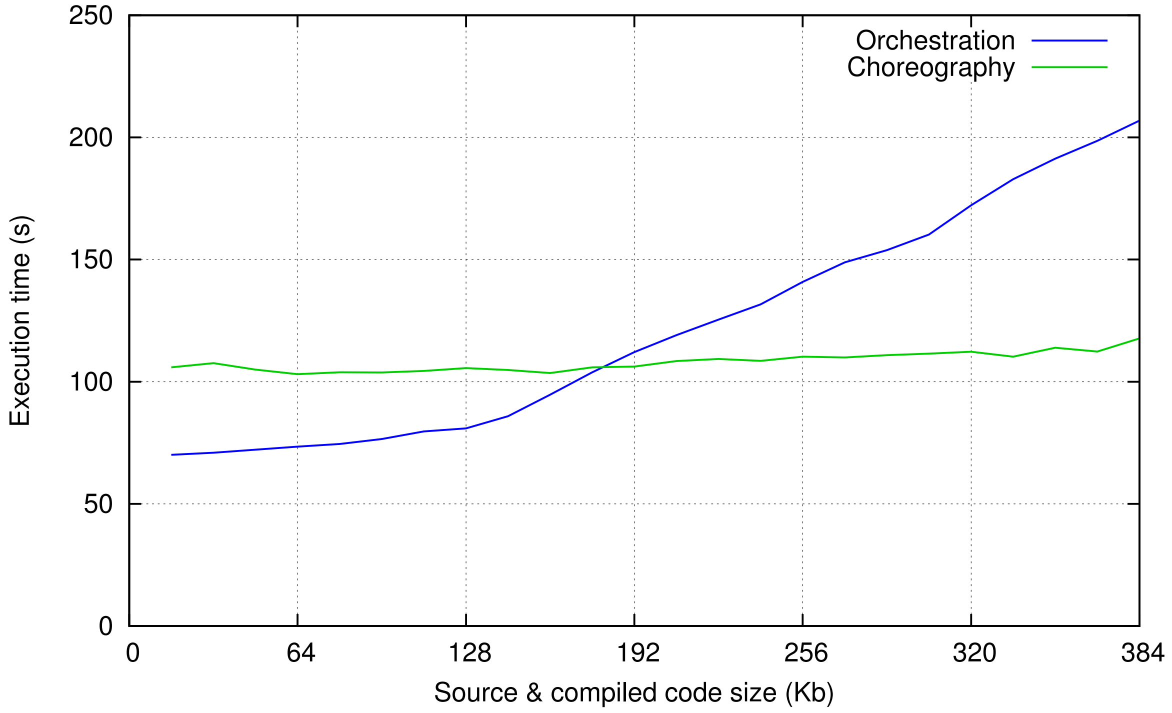


Figure 7.34: Student marks workflow — execution time vs. code size (32 nodes)

each service request sends the same amount of code, and for choreography, because the number of code objects distributed among machines should be independent of their size. To verify this, we ran an experiment involving a fixed number of nodes and test cases, while varying the size of the source code and compiled code associated with each student's submission. In each case, the source and compiled code were of equal size, though as discussed in Section 7.5.1, there is a 33% overhead for the compiled binary code, due to Base64 encoding in the SOAP message. As with the first experiment, this experiment used a computation time of one second per `runTest` call, with 64 students and 64 test cases.

The results for data transfer are shown in Figure 7.33. As was expected, the increase in data transfer was linear in both cases, with the same ratio of communication between orchestration and choreography. For very small code sizes, choreography actually resulted in greater amounts of data being passed between machines, due to the overheads of control messages. However, once the code size was 32 Kb, the two were almost even. For larger code sizes, the benefit of choreography in terms of the amount of communication is very significant.

With this experiment, we also observed improvements in execution time for choreography vs. orchestration, as shown in Figure 7.34. In all of our previous experiments, choreography took longer than orchestration, due to scheduling overheads. However, the purpose of choreography is to reduce execution time for data-intensive workflows running in low-bandwidth environments. With this experiment, even on the 100 Megabit network connecting the nodes in the cluster, we were able to observe significant improvements in this respect.

Looking at Figure 7.34, it can be seen that the total execution time varies only slightly for choreography, but increases dramatically for orchestration above the 144 Kb mark. At this point, with orchestration, the client node saturates its outgoing bandwidth, and communication becomes the dominant factor in execution time. With choreography, there is much less data being passed around, and the total communication is spread across all machines. In this situation, choreography offers very substantial performance benefits — taking only 57% of the time of orchestration at the 384 Kb mark.

7.6 Internal computation

A key goal of this project was to support the inclusion of application logic and data processing within a workflow. Whereas previous sections have focused on parallel service invocation and optimisation of data transfers, we now turn our attention to the performance of internal computation.

When considering this aspect of performance, it is important to do so in light of what we have achieved above. Both ELC and our XQuery implementation support automatic parallelisation, and make it easy to seamlessly coordinate large numbers of service invocations across multiple machines. Complex details are abstracted away from the programmer, enabling them to focus on what their workflow does, not how it does it. These benefits,

Language	Implementation/version
C	GCC 4.1.2; optimisations enabled (-O3)
Java	Sun JDK 1.6.0_14
Haskell	GHC 6.12.1; optimisations enabled (-O3)
Python	2.6.4
Perl	5.10.1
Interpreted JavaScript	Mozilla SpiderMonkey 1.8.0rc1
Compiled JavaScript	Google V8 2.1.0.1

Table 7.2: Languages used for computation performance tests

however, don't come for free — they depend on complex runtime support mechanisms which introduce certain overheads to execution of code within a workflow.

Our intention has been to support *lightweight* computation within a workflow — simple application logic and data manipulation which augments the functionality of existing tasks, but is not performance critical. Compute-intensive parts of a workflow should be implemented as external services using compiled, sequential programming languages such as C and Java. Our intention was not to compete with these languages, but instead to provide a level of performance comparable to that of scripting languages. In this section, we present results that show we have achieved this goal.

7.6.1 ELC

To measure the execution speed of ELC, we developed a small collection of benchmark programs designed to test different aspects of runtime performance. Each program was implemented in seven different languages: C, Java, Python, Perl, JavaScript, Haskell, and ELC. These languages were chosen based on popularity. Although Haskell is not widely used in industry, it was chosen to allow us to benchmark against another functional language. The results presented here compare the performance of ELC with each of these languages.

Technically, it is not actually the languages themselves we are measuring, but rather specific *implementations* of these languages. Different implementations of a language can have significantly different performance characteristics. This is especially true of JavaScript, where browser developers regularly compete with each other to improve performance. Alternative implementations of ELC are also possible, and it is really NReduce itself that we are comparing with these other language implementations. Table 7.2 lists the implementations of each language that we used for our tests. Those used for C, Java, and Haskell are compilers, while Python and Perl use interpreters.

JavaScript is a special case. Traditionally it has always been interpreted, however new implementations have recently emerged which use JIT (just in time) compilation to achieve dramatic performance improvements [63]. These are only available in some browsers however, and a significant portion of Internet users still run browsers containing JavaScript interpreters. We therefore ran our benchmarks with two different JavaScript engines — an interpreter (SpiderMonkey; used in Mozilla Firefox until mid-2009), and a JIT-compilation engine (V8; used in Google Chrome).

The benchmarks used for our performance tests are as follows:

- **bintree.** Creates a binary tree of depth n, then counts the total number of nodes in it. This benchmark is used to measure the performance of memory allocation and garbage collection. It was chosen because it involves large amounts of memory allocation and relatively little computation. It allows the amount of memory allocation to be easily varied over a very wide range, since the memory usage increases exponentially with respect to n.

- **mandelbrot.** Renders an area of the Mandelbrot set [215] in ASCII art, where the width and height depend on the parameter n. This benchmark measures floating point performance. It was chosen because it spends most of its time performing floating point computations, thereby isolating the effects of other aspects such as function calls and memory allocation.

- **matmult.** Creates two $n \times n$ matrices, multiplies them together, and prints out the results. This benchmark measures the performance of arrays. It was chosen because it performs a large number of indexing operations on arrays, thus testing how well the optimised list representation described in Section 4.6 works in comparison to the native array support of other languages.

- **nfib**. Computes all members of the Fibonacci sequence up to n, using a deliberately inefficient method in which $\text{nfib}(n)$ is computed by evaluating the expression $\text{nfib}\,(n-1) + \text{nfib}\,(n-2)$. This benchmark tests function call performance. It was chosen because it generates a very large number of function calls which each perform only a small amount of computation. The results for this benchmark therefore reflect the cost of function calls in each language implementation.

- **mergesort** and **quicksort**. Both create a list of n strings, based on taking an incrementing number sequence, converting each number to a string, and reversing the digits. These strings are then sorted using either the merge sort[4] or quick sort algorithm, and the results are printed. These benchmarks test the performance of memory allocation, garbage collection, and computation. They were chosen because they contain a mixture of these aspects, and therefore give an overall indication of how well algorithms that use lots of memory and computation perform in each language.

The results given in the following sections are the average times observed over a series of ten runs.

[4]The algorithm used is a slightly optimised version of the one shown in Figure 3.9 on page 116.

7.6.1.1 bintree

The binary tree benchmark, when run with $n = 22$, involves construction of almost 4.2 million objects, and thus reflects the speed of memory allocation and garbage collection in each language. As can be seen in Table 7.3, ELC is almost twice as fast as the interpreted languages on this test, but is slower than all of the compiled languages. Java and compiled JavaScript both run approximately twice as fast as ELC, which we attribute to both better code generation and garbage collection. C completes very quickly, as it does not use garbage collection at all. Haskell is just as fast as C on this test, although we suspect this is due to the compiler using fusion [65] to merge the construction and traversal logic together, eliminating allocation (and hence garbage collection) of the tree.

	User time (s)							
n	ELC	C	Java	Haskell	Compiled JavaScript	Interpreted JavaScript	Python	Perl
16	0.1	0.0	0.2	0.0	0.1	0.2	0.3	0.4
18	0.2	0.0	0.3	0.0	0.2	0.8	1.2	1.7
20	1.7	0.1	0.6	0.1	0.7	3.3	6.9	6.9
22	6.9	0.6	3.7	0.5	3.4	13.4	54.5	27.0

Table 7.3: Results for bintree

One downside of this benchmark in terms of measuring garbage collection performance is that it uses a large number of long-lived objects. It has been observed by others [261] that in most programs, the majority of objects live for only a short period of time. NReduce uses generational garbage collection, which works with this assumption, and makes collection of short-lived objects much cheaper than long-lived ones. Because this benchmark uses a large number of long-lived objects — the tree must be fully constructed before its nodes are counted — garbage collection is more expensive for this than for many other programs. For $n = 22$, the ELC version spends roughly 54% of its time in garbage collection.

This benchmark does demonstrate however that even with the additional cost of long-lived objects, NReduce still performs significantly better than the interpreted languages. We believe this to be primarily attributable to the cost of object creation and garbage collection in our implementation compared to these other languages, although the speed of the generated code vs. interpretation may also be a factor.

7.6.1.2 mandelbrot

The mandelbrot benchmark is designed to test the performance of each language on numerically intensive code. The program involves only a small number of function calls, and virtually no memory allocation; almost all of the time is spent in loops computing values based on complex numbers, represented as pairs of variables. The results are shown in Table 7.4.

	User time (s)							
n	ELC	C	Java	Haskell	Compiled JavaScript	Interpreted JavaScript	Python	Perl
32	0.5	0.0	0.2	0.1	0.3	3.4	5.3	4.2
64	1.9	0.1	0.4	0.5	1.2	13.9	21.1	16.5
96	4.3	0.3	0.6	1.1	2.7	31.7	46.9	37.6
128	7.6	0.5	1.1	1.9	4.7	54.9	83.1	65.8

Table 7.4: Results for mandelbrot

What is perhaps surprising in this benchmark is the degree of difference between the languages. There is a factor of more than 100 between the performance of the fastest language (C) and the slowest language (Python) on this test. The poor performance of the scripting languages can primarily be explained by interpretive overheads, and this is where the bytecode compilation performed by NReduce really makes a major difference.

Despite the use of compiled code however, NReduce is still significantly slower than the compiled languages on this test. We believe that the most likely reason for this is poor use of registers in the generated code; each numeric operation loads the necessary values from memory, performs the operation, and stores the result back in memory. It is likely that Java, C, and Haskell avoid these unnecessary transfers in many cases and leave values in the floating point registers for subsequent operations to act upon.

Even with the aforementioned performance limitations of our implementation, the results are still pleasing when considering that NReduce was never intended for numerically intensive computation, and our original plan was to only use an interpreter for running ELC code. The fact that its performance is still a factor of seven times greater than the fastest of the interpreted languages illustrates that it is very competitive in this area, and the performance is quite sufficient for the lightweight computation required for workflows.

7.6.1.3 matmult

The purpose of the matrix multiplication test is to measure the performance of arrays. All of the languages used in the test support arrays, however ELC does so implicitly rather than explicitly. A programmer working with ELC operates on a list in an abstract manner, and is unaware of whether it is physically represented using cons cells, arrays, or a combination of both. The concrete representation is purely an implementation detail, hidden by the virtual machine, as described in Section 4.6. By using the list functions provided in the prelude module, such as `len` and `item`, indexed access to a list can be performed in O(1) time in the case where it happens to be stored as an array.

The results from this experiment are shown in Table 7.5. As expected, C, Java, and compiled JavaScript vastly outperformed the interpreted languages on this test. However, the ELC version was slower than expected. The difference can be explained by the fact that even though array access is O(1), each access carries some additional constant overhead involving dynamic type checking. The `item` function also has to deal with the

	User time (s)								
n	ELC	C	Java	Haskell (List)	Haskell (Array)	Compiled JavaScript	Interpreted JavaScript	Python	Perl
100	0.8	0.0	0.2	0.6	0.5	0.1	0.7	1.3	1.3
200	5.7	0.1	0.3	31.9	5.1	0.5	5.6	9.9	11.0
300	19.2	0.7	0.8	200.9	24.4	1.7	18.5	33.2	34.7
400	45.5	1.7	1.9	651.6	82.2	5.7	44.1	77.4	84.5

Table 7.5: Results for matmult

fact that the argument may actually be a linked list of arrays and/or cons cells, so the access mechanisms are considerably more complex than simply dereferencing a pointer and accessing memory at a particular offset. However, even with these overheads, its speed is still comparable to the interpreted languages, and is definitely much better than would be seen if all cons lists were stored as linked lists and accessed in an O(n) fashion.

For this test, we implemented two different versions of the program in Haskell — one which uses lists, and another which uses arrays. The list-based version was identical to the code for the ELC implementation, except for syntactic differences. Because Haskell represents all lists using cons cells, this version exhibited extremely poor performance, because each indexed access incurs an O(n) cost. The array-based version performed much better, though it was still slower than ELC. This version was an entirely separate implementation from the list-based one, since using arrays in Haskell requires a different style of programming to using lists. In this context, ELC has the advantage of allowing the programmer to take advantage of arrays by simply using the built-in list functions, instead of having to use a separate and conceptually different set of operations for arrays.

7.6.1.4 nfib

The nfib benchmark tests the performance of function calls. Because each call does very little computation, the majority of the execution time is taken up by the function call/return mechanism. Table 7.6 shows the results.

	User time (s)							
n	ELC	C	Java	Haskell	Compiled JavaScript	Interpreted JavaScript	Python	Perl
30	1.1	0.0	0.2	0.1	0.2	3.4	5.4	10.3
32	2.9	0.1	0.2	0.2	0.3	8.8	13.8	26.9
34	7.5	0.2	0.4	0.5	0.9	22.8	37.5	70.3
36	19.6	0.4	0.9	1.2	2.2	59.5	100.2	184.0

Table 7.6: Results for nfib

This test is a particularly important one for evaluating the efficiency of NReduce, because of our design decision to treat threads and function calls as the same thing. In most languages, thread creation is much more expensive than a function call, since in order to create a thread, the program must make a system call which creates a kernel-level object to represent the thread. In contrast, our implementation represents threads/function calls as frames, which are similar to stack frames in imperative languages, but arranged as a graph. Had this aspect of our execution mechanism not been efficient, the associated performance limitations would have been a show-stopper, since one of our main goals is to support efficient computation within workflows.

Fortunately, this has turned out not to be the case, and the results show that function calls do perform reasonably well. Although the performance is significantly less than that of the compiled languages (which was to be expected), we are still able to achieve at least three times times the performance on this benchmark than the interpreted languages — and almost an order of magnitude faster than Perl.

The differences in function call cost between ELC and C can be accounted for as follows:

- Frames are allocated in a heap, rather than a stack. This reduces cache locality, because the memory associated with a given frame is less likely to be in the cache when the frame is activated.
- Arguments to a function call must be copied into the area of memory used by the frame, instead of accessed directly from the caller's memory. C compilers use the latter approach, but this is not possible in our case since we need to be able to have the caller continue on with other work while the called function is in progress, which may cause the caller to modify the stack positions in which the arguments were previously stored.
- When the result of a function call will not be used immediately, a cell must be allocated on the heap to hold its return value once the call has completed. This incurs costs for both memory allocation and garbage collection.
- Additional instructions must be executed on both function call and return to update the linked list of runnable frames maintained by the task, as well as the blocked list of the called frame, as discussed in Section 4.4.1.
- The C version uses ints, while the ELC version uses doubles (the only numeric type supported). Modifying the C version to use doubles showed an increase of around 70% in execution time, suggesting that this is also a factor.

These costs are mostly inherent in the execution model we have chosen. While it may be possible to squeeze some additional performance out by further optimisations to the call/return sequence, it appears that this is about as good as it is likely to get, without changing the execution strategy. Some other parallel functional language implementations, such as GHC [163], are based on the more traditional stack model that C uses, and utilise multiple threads to handle parallelism. This may be a solution to increasing performance, however it would significantly complicate the implementation of blocking on service requests, as well as automatic parallelisation.

Overall though, we believe the performance on this benchmark reflects reasonably good performance, and is certainly adequate for the intended applications. In particular, the fact that it is significantly faster than the interpreted languages, which don't support automatic parallelisation at all, means that it is certainly viable for lightweight computation.

7.6.1.5 mergesort and quicksort

The sorting benchmarks test a mixture of both the memory and computation-intensive aspects of performance. For each language, we implemented both of the programs in a side effect free manner, in order to give a fair comparison with ELC. Both programs took reasonably similar times in each language, with quicksort being slightly faster than mergesort. The results for the tests are given in Tables 7.7 and 7.8.

	User time (s)							
n	ELC	C	Java	Haskell	Compiled JavaScript	Interpreted JavaScript	Python	Perl
100000	3.4	0.2	3.4	1.6	1.2	3.5	4.3	11.6
200000	8.6	0.4	4.9	3.7	2.4	7.4	9.0	24.8
300000	13.9	0.7	7.4	5.8	3.7	11.3	13.7	39.8
400000	20.5	1.0	8.6	8.5	5.2	15.4	19.0	53.5

Table 7.7: Results for mergesort

	User time (s)							
n	ELC	C	Java	Haskell	Compiled JavaScript	Interpreted JavaScript	Python	Perl
100000	2.8	0.2	2.0	0.9	1.0	3.4	3.1	7.3
200000	7.5	0.4	3.4	2.0	2.1	7.1	6.5	16.2
300000	13.9	0.6	4.8	3.3	3.2	10.9	10.2	25.7
400000	19.1	0.9	6.5	4.8	4.4	14.9	13.8	36.4

Table 7.8: Results for quicksort

Again, ELC was competitive with the interpreted languages, but not to the extent that it was on other benchmarks like bintree and nfib. We believe this can be attributed to the fact that the ELC version uses linked lists, which require a new allocation operation for each list item — even if those lists are later converted to arrays internally. In contrast, the implementations in the other languages make exclusive use of arrays, which are allocated once and then filled. Since the ELC version creates a much larger number of objects, it spends a lot more of its time performing allocation and garbage collection. NReduce spends roughly 50% of its time performing garbage collection in these tests.

7.6.1.6 Discussion

As mentioned in Section 4.1, our performance goals for NReduce were to execute code at a speed competitive with mainstream interpreted scripting languages. The results for the above experiments show that we have achieved this goal. Scripting languages like Perl, Python, and JavaScript are used in a wide range of scenarios in which ease of use is an important concern, and the performance of their interpreters is generally considered adequate for simple computation. Since the computation aspects of ELC are designed for similar types of usage, we believe the performance we have achieved to be quite sufficient.

NReduce does make limited use of native code generation in order to improve performance, but does not perform many of the optimisations that are traditionally found in compilers, since this was not the focus of our project. There were substantial differences between the results for compiled languages and those for ELC, which we believe can be accounted for by the following factors:

- Every expression is considered a candidate for parallel evaluation, and thus incurs overheads relating to spark management.
- To enable multiple function calls to be active at a time, the frames are represented as a graph, instead of a stack. This requires more work to update on each function call and return, as discussed in Section 4.4.1.
- Since ELC is a dynamically typed language, a large number of type checks must be performed at runtime.
- The generated code makes poor use of registers, loading and saving values from memory on each bytecode instruction.
- All pointers are represented using two words, as discussed in Section 4.2.4, which makes them more expensive to copy than the traditional single-word pointer representation used by most other languages.

We believe that these aspects could all be improved upon, mostly by applying standard compiler optimisation techniques such as type inference and register allocation. However, automatic parallelisation comes with an inherent cost that is impossible to completely avoid, at least using currently known techniques. Although future improvements to computation speed would be nice, the current performance is adequate for lightweight processing of the type normally implemented in scripting languages.

What we have demonstrated with our implementation is that it is possible to combine fine-grained computation with the automated detection of parallelism and dynamic scheduling required for workflows. Where previous workflow engines have been designed only with coarse-grained, external computation in mind, we have demonstrated that it is possible to achieve these goals while still supporting fine-grained, internal computation. We have achieved this by implementing our workflow engine in a manner which is much more closely aligned with traditional techniques for constructing programming language implementations than has previously been taken for other workflow engines.

7.6.2 XQuery

XQuery is designed for querying and manipulating XML data. The most important performance aspects of an XQuery implementation are the document sizes that it can handle, and the speed at which it can locate nodes in the document tree based on specified criteria. Computational facilities such as arithmetic operators are mainly used in an ancillary manner, as part of filter predicates and for computing values of attributes and text nodes in constructed content. We therefore focus on the data querying aspects in our evaluation.

XQuery processors fall into two main categories: databases and standalone processors. Implementations designed for database queries use large, persistent, indexed data stores, and scale to very large document sizes. In contrast, standalone processors operate on an in-memory representation of the document, and do not have access to pre-computed indices. Standalone processors are less scalable, as they are limited to documents which can fit in memory. Since our implementation fits into this latter category, we chose to benchmark it against another standalone processor: version 9.0 of Saxon-B [177], an open source, standalone XQuery processor written in Java.

The tests we carried out were taken from XMark [264], an XQuery benchmark suite used widely in the literature [7, 216]. It consists of twenty different queries, each of which exercises a different aspect of the language. The queries are designed to run against input files conforming to a particular schema designed for a fictitious online auction site. A program called `xmlgen` is used to generate these files; it takes a *factor* as input, and generates a document whose size is a multiple of the factor. For a factor of 1, `xmlgen` produces an XML file of 111 Mb, containing approximately 781,000 elements.

In running these tests, we sought to determine the scalability of each implementation. To ensure the results incorporated both the sizes of input documents that could be handled and the speed of query execution, we imposed a limit of 30 seconds and 512 Mb of memory on all runs. We ran each query multiple times against increasing document sizes, and recorded the largest size for which the query successfully completed within the time and memory limits. All queries were run except for q19; this uses the `order by` clause, which is not supported by our implementation. Details of each query are given in [264]. The results are shown in Table 7.9.

The differences in performance between queries in our implementation can be accounted for as follows: Queries 1 – 5, 13, and 15 – 20 run the fastest, because they involve a single tree traversal using child steps, which can be evaluated relatively quickly. Queries 6, 7, and 14 use the `descendant-or-self` axis, which we implement by constructing a temporary list containing the node and all of its descendants, and then evaluating the path expression for each element in this list. These queries run more slowly due to the costs of allocating this temporary list, which is not necessary for the child axis, which is able to use the existing list that is already present for each node. Queries 8 – 12 involve joins expressed as nested `for` loops, which we do not optimise; these queries therefore require a much larger number of iterations than the others.

The results in Table 7.9 show very large performance differences between Saxon and our

Query	XQuery+NReduce (ELC parser)			Saxon			Ratio of factors
	Factor	File size	Limited by	Factor	File size	Limited by	
q1	0.128	14.3 Mb	Time	1.190	132.6 Mb	Memory	9.3
q2	0.119	13.2 Mb	Time	1.158	129.1 Mb	Memory	9.7
q3	0.110	12.4 Mb	Time	1.171	130.4 Mb	Memory	10.6
q4	0.099	11 Mb	Time	1.171	130.4 Mb	Memory	11.8
q5	0.128	14.3 Mb	Time	1.184	131.9 Mb	Memory	9.3
q6	0.056	6.2 Mb	Time	1.178	131.2 Mb	Memory	21.0
q7	0.047	5.3 Mb	Time	1.171	130.4 Mb	Memory	24.9
q8	0.039	4.4 Mb	Time	0.206	23 Mb	Time	5.3
q9	0.036	4 Mb	Time	0.189	21.2 Mb	Time	5.3
q10	0.062	6.8 Mb	Time	0.499	55.3 Mb	Time	8.0
q11	0.031	3.5 Mb	Time	0.218	24.3 Mb	Time	7.0
q12	0.055	6.1 Mb	Time	0.352	39.2 Mb	Time	6.4
q13	0.128	14.3 Mb	Time	1.165	129.8 Mb	Memory	9.1
q14	0.058	6.4 Mb	Time	1.178	131.2 Mb	Memory	20.3
q15	0.121	13.6 Mb	Time	1.158	129.1 Mb	Memory	9.6
q16	0.117	13 Mb	Time	1.190	132.6 Mb	Memory	10.2
q17	0.115	12.7 Mb	Time	1.165	129.8 Mb	Memory	10.1
q18	0.120	13.4 Mb	Time	1.178	131.2 Mb	Memory	9.8
q20	0.116	13 Mb	Time	1.184	131.9 Mb	Memory	10.2

Table 7.9: XMark benchmarks with XML parser implemented in ELC

implementation; on average, there is just over an order of magnitude difference between the two. When investigating the reasons why our implementation was so slow, we discovered that the bottleneck was in the XML parser, which is implemented in ELC. The vast majority of bytecode instructions were in the parser code, and 85% of memory allocations were temporary objects created during parsing which were discarded once construction of the document tree was completed. This contributed to an average of 60% of execution time being spent on garbage collection.

To remove this bottleneck, we tried a C implementation of the XML parser, using libxml [313] to parse the file, and then traversing the node tree it produced to construct a tree of ELC XML node objects in the same format as used by the rest of the code. After re-running the tests, we found a dramatic improvement in performance, with our engine being able to handle much larger documents. The results obtained using this approach are shown in Table 7.10, which show that we are able to achieve scalability within a third of that of Saxon in the best case, and just under a quarter on average.

Saxon is faster because it contains a number of optimisations, described in [178] and [180], that are not present in our implementation. One of these is *pipelining*, which reduces the amount of temporary memory allocation needed during a query by passing results directly from producers to consumers, without placing them in temporary data structures. This makes a significant difference for queries 6, 7, and 14 which, as discussed

Query	XQuery+NReduce (C parser)			Saxon			Ratio of factors
	Factor	File size	Limited by	Factor	File size	Limited by	
q1	0.432	48.1 Mb	Memory	1.190	132.6 Mb	Memory	2.8
q2	0.432	48.1 Mb	Memory	1.158	129.1 Mb	Memory	2.7
q3	0.432	48.1 Mb	Memory	1.171	130.4 Mb	Memory	2.7
q4	0.432	48.1 Mb	Memory	1.171	130.4 Mb	Memory	2.7
q5	0.432	48.1 Mb	Memory	1.184	131.9 Mb	Memory	2.7
q6	0.136	15.1 Mb	Time	1.178	131.2 Mb	Memory	8.7
q7	0.098	10.9 Mb	Time	1.171	130.4 Mb	Memory	11.9
q8	0.049	5.4 Mb	Time	0.206	23 Mb	Time	4.2
q9	0.046	5.2 Mb	Time	0.189	21.2 Mb	Time	4.1
q10	0.110	12.4 Mb	Time	0.499	55.3 Mb	Time	4.5
q11	0.035	3.9 Mb	Time	0.218	24.3 Mb	Time	6.2
q12	0.074	8.2 Mb	Time	0.352	39.2 Mb	Time	4.8
q13	0.432	48.1 Mb	Memory	1.165	129.8 Mb	Memory	2.7
q14	0.161	17.8 Mb	Time	1.178	131.2 Mb	Memory	7.3
q15	0.432	48.1 Mb	Memory	1.158	129.1 Mb	Memory	2.7
q16	0.432	48.1 Mb	Memory	1.190	132.6 Mb	Memory	2.8
q17	0.432	48.1 Mb	Memory	1.165	129.8 Mb	Memory	2.7
q18	0.432	48.1 Mb	Memory	1.178	131.2 Mb	Memory	2.7
q20	0.432	48.1 Mb	Memory	1.184	131.9 Mb	Memory	2.7

Table 7.10: XMark benchmarks with XML parser implemented in C

above, require the allocation of large temporary lists in our implementation. Another optimisation Saxon performs is query rewriting, where certain expressions are transformed into equivalent expressions which can be executed more efficiently, sometimes based on statically computed information about parts of the expression tree.

We believe that the main reason for the differences between the performance of Saxon and our own XQuery implementation is the tree representations used. We represent each XML node as a separate object, stored as a list, and access fields by indexing into specific positions within these lists. This incurs the costs of memory allocation and garbage collection for each node, as well as runtime type and bounds checks when accessing an object. In contrast, Saxon uses a representation called TinyTree, in which an entire tree of nodes is represented using six arrays of integers, where each array contains all the values for a specific property. Node names are stored using integers, which act as indices into a name table. This enables them to be efficiently compared using integer comparison, instead of slower string comparison operations that we use instead. Results given in [178] compare Saxon's performance using TinyTree against a linked node representation similar to our own, and report a six times performance improvement for TinyTree with a 100 Mb version of the XMark data set.

We find the performance results obtained for our XQuery implementation, particularly with the C parser, to be encouraging. Our implementation is intended as a proof of concept, and has had considerably less effort invested into optimisation relative to Saxon. We believe that with sufficient additional engineering, the performance could be brought closer to that of Saxon, however the tree representation that our implementation is based on does carry limitations relative to TinyTree, as discussed above. Our query processing logic is designed for small amounts of lightweight computation used to manipulate intermediate data exchanged between services in a workflow, and the performance we have achieved is quite adequate for this purpose. As we have demonstrated here, it is possible to work with moderately large amounts of XML data with a reasonable amount of memory using our system.

7.7 Summary

In this chapter, we reported on a wide range of experiments designed to test different performance characteristics of our system. These included measurements of scalability, granularity, speedup for various types of parallelism, reduction of data transfer between machines, and internal computation. Experiments were carried out using workflows written directly in ELC, as well as workflows written in XQuery which were compiled into ELC. The results we have presented demonstrate the following achievements of our work:

- NReduce is able to handle very large-scale workflows, involving up to two million tasks. High levels of speedup are achievable for task granularities of one second or more for choreography, and 32 milliseconds or more for orchestration.

- Our system is successful in automatically detecting all of the parallelism available within a workflow, and managing the parallel invocation of multiple services. Parallelism is completely abstracted away from the programmer.

- Our generic approach to detecting parallelism works well for a range of common forms of parallel processing. All of these forms can be expressed in terms of user code, without needing to be supported explicitly as native language constructs. The results presented in this chapter show that our system is capable of achieving almost the maximum possible speedup for each style of parallel processing that we tested.

- The distributed nature of our virtual machine allows it to successfully achieve choreography of services, greatly reducing the amount of bandwidth used when executing a workflow. This is achieved by transferring data directly between service hosts, instead of via a central client. Importantly, this mechanism is completely automatic, and requires no explicit direction from the programmer.

- XQuery workflows, by virtue of being supported via translation to ELC, are able to take advantage of the parallel execution and choreography facilities of the virtual machine, and obtain the same performance benefits as workflows written directly in ELC.

- Perhaps most significantly, we have achieved the above *in conjunction with* efficient execution of fine-grained computation. The performance we have achieved for internal computation rivals that of widely-used interpreted scripting languages, making it feasible for users to include significant amounts of application logic and data manipulation within their workflows.

Taken together, the results presented in this chapter demonstrate that we have succeeded in achieving the performance levels necessary to support the types of workflows we have targeted. Although there are several areas in which performance tuning could give further improvements, our work is successful as a proof of concept. We have shown that the design philosophy on which our virtual machine has been constructed is sound, such that our ideas can be used in the construction of future workflow engines. We hope that the results and analysis presented in this chapter will help to convince the workflow community that integrating fine-grained internal computation with the coarse-grained coordination facilities of workflow languages is both useful and achievable.

Chapter 8

Summary and Conclusions

This thesis has explored part of the design space of workflow languages, concerned with achieving a greater degree of expressiveness than is possible with existing languages. We have achieved this by demonstrating how existing techniques from the functional programming literature may be applied in the construction of a workflow language, in the process realising a feature set that is supported by neither existing functional languages nor workflow systems. Our work has been driven by the following observations from the literature:

- There is significant demand for powerful workflow languages that enable distributed scripting of software components within a service oriented architecture.
- Existing workflow languages have only limited capabilities compared to mainstream imperative and functional programming languages.
- Though powerful, existing general-purpose programming languages do not fully address the requirements of workflows, specifically with regards to parallel invocation of external services.
- There is a major intellectual and cultural division between the worlds of workflow/distributed computing systems and (functional) programming language theory, resulting in very little cross-fertilisation of ideas.

8.1 Summary

We began by examining the requirements of data-oriented workflows, and the capabilities and limitations of existing workflow languages. The almost exclusive focus of these languages on coordination rather than computation limits the range of logic that can be expressed in a workflow, often requiring users to augment the functionality of services with additional tasks written in separate, imperative scripting languages. In theory, functional programming languages are ideally suited to providing both the coordination and computation facilities required for workflows, as they are based on a purely data-oriented model, and support arbitrary computation and data manipulation. In practice however, existing functional languages lack support for automatic parallelisation and the ability to communicate with network services in the context of pure expressions.

Our work is the first to develop a programming model which fully integrates ideas from functional programming and workflows. We developed a language called ELC, which extends lambda calculus with a small set of data types and built-in functions necessary for the practical implementation of workflows. In particular, it provides a purely data-oriented abstraction of network connections, which allows services to be accessed within the context of pure expressions, and automatically invoked in parallel where possible. Our motivation for developing ELC was to demonstrate the merits of using the well-established model of lambda calculus as the basis for a workflow engine, rather than re-inventing the wheel by attempting to create an entirely new language from scratch.

We demonstrated the practical application of this model by designing and implementing the NReduce virtual machine, which executes workflows written in ELC. This virtual machine is based on established techniques for implementing parallel functional languages, but includes several innovations that are specific to workflows — in particular, the ways in which parallelism, network I/O, and choreography are managed. Our virtual machine has been extensively benchmarked with a range of different workflow types, and shown to achieve high levels of performance and scalability. These results were achieved through extensive efforts invested into tuning and optimisation applied to all parts of the implementation, including compilation, management of parallelism, load distribution, and distributed memory management.

One of the most important features of our language implementation is that it automatically detects opportunities for parallel evaluation, without requiring any explicit direction from the programmer. This is feasible due to our focus on *external parallelism* involving coarse-grained remote service calls, in contrast to the *internal parallelism* that has been the focus of previous work on parallel functional programming. It is the coarse-grained nature of tasks invoked within workflows that makes automatic parallelisation a realistic option, since the costs of extracting as much parallelism as possible from a workflow remain small compared to the amount of time spent waiting for remotely executed service operations to complete.

To reduce bandwidth requirements, the virtual machine can perform service choreography when deployed in a distributed manner across all the machines hosting the services used within a workflow. In this mode of operation, data transferred between different services is passed directly between the nodes of the virtual machine, instead of being routed through a central client node, as occurs in orchestration. Implementing this distributed execution mode involved development of an object-based distributed shared memory system, a peer-to-peer message passing and process management infrastructure, and effective load balancing schemes similar to those used for other parallel graph reduction machines but explicitly customised to take into account complexities associated with external service access.

We also developed an implementation of XQuery, extended with support for web services, to allow workflows to be defined in a manner which makes it easy for programmers to deal with XML data exchanged with web services. XQuery is a highly appropriate choice for this task, since it is a side effect free functional language, provides excellent support for XML processing, and is a widely recognised standard. The key innovation of our implementation is that rather than implementing XQuery directly in a low-level language like C, we instead compile it into ELC, making it possible to leverage all of the benefits of our virtual machine for execution. This means that we are able to automatically parallelise XQuery code, and use it in conjunction with the choreography mechanisms described above.

The performance results we obtained in our experiments indicate that our approach is successful at exploiting all of the parallelism available in workflows, and delivering high levels of speedup. Additionally, the performance of internal computation is comparable to that of several widely used interpreted scripting languages. These results demonstrate that our goal of supporting efficient coordination and computation within the same workflow language has been met.

8.2 Review of case study: Image similarity search

In Section 1.1, we began by considering an application scenario involving the generation of an index to support image similarity queries. This example illustrated a number of requirements which are important for data processing workflows of the form often used in scientific scenarios. Our goal was to develop a programming model and implementation capable of supporting the style of processing embodied in this use case. We successfully achieved this goal, and in Section 3.7 gave a detailed explanation of how this workflow could be implemented using our model.

Let us consider the requirements listed in Section 1.2, and review how we have gone about supporting them.

- **Remote service access.** Our use of a parallel functional programming model necessitated a purely data-oriented abstraction for accessing services. In addition, the need to support many different types of services required that programmers are given the opportunity to define arbitrary encoding mechanisms for exchanging data with services. Section 3.5.8 presented the functionality provided by our programming model for accessing services, which exposes network connections as a function which transforms requests into responses. Section 4.5 discussed how multiple concurrent connections are handled within the virtual machine. Section 3.7.2 examined these service access mechanisms in the context of the example workflow. For workflows based on web services, our XQuery implementation can be used to automatically generate service access functions.

- **Parallelism.** As with many other workflow languages, our programming model supports automatic detection of parallelism, relieving the programmer of the responsibility of managing parallel execution manually. While this is straightforward for the dataflow models upon which existing workflow languages are based, the introduction of sophisticated control flow facilities like recursion and higher-order functions necessitates a more flexible approach. We implement automated detection of parallelism in a very general manner which makes it applicable to a many different styles of parallelism, as demonstrated in Section 7.3. The example workflow is able to take advantage of data parallelism, because it relies on looping constructs such as `map` and `filter` in which each function application is independent of the others.

- **Data structures.** We demonstrated that supporting structured data within a workflow can be done very simply by using the tried and tested idea of cons cells [221]. These can be used to represent arbitrary data structures, including pairs, tuples, lists, and trees, and their construction and manipulation requires only three primitive functions — `cons`, `head`, and `tail`. Section 3.7 discussed how cons cells could be used to represent the tuples required by the example workflow. Programmers can easily implement additional abstractions using standard list manipulation functions, several of which are provided by our language implementation as library functions, for purposes of convenience. Our XQuery implementation builds on these mechanisms to represent and manipulate XML data.

- **Internal computation.** Since our programming model is based on lambda calculus, it supports the expression of arbitrary computation. Any intermediate computations that need to be performed on data received from services can thus be expressed directly in the workflow language, instead of as external tasks written in separate scripting languages, as required by other workflow systems. Part of the example workflow involves sorting a list of images; the sorting algorithm can be implemented in terms of our programming model, and thus does not need to be provided as a language primitive. Any other algorithm can be implemented similarly.

- **Control flow.** In addition to being necessary for implementing most forms of computation, control flow constructs are needed for the overall coordination of workflows. Many different forms of control flow can be expressed in terms of lambda calculus, often as higher-order functions which accept expressions such as loop bodies as parameters. The example workflow requires nested loops to compare every image with every other image, and higher-order functions to permit the ordering criteria to be specified when sorting, and the selection function to be specified when filtering lists of images.

In working towards these requirements, we have brought together concepts from both functional languages and workflow languages. At a theoretical level, functional languages are very similar to data-oriented workflow languages, in that they are based on a purely side effect free model of computation in which data dependencies between operations are explicit, and can thus be used to reliably detect opportunities for parallel evaluation. Functional languages offer several benefits over existing workflow languages, including support for higher-order functions, which allow certain classes of control flow structures to be defined in user code, a concise syntax, which is convenient for programmers to use, and standard built-in functions for data manipulation and list processing.

Our work has demonstrated that it is possible to provide all of the features necessary for data-oriented workflows using a functional language, instead of a language which is designed solely for coordination purposes. We hope that the explanations given in this thesis will help foster an understanding of the notion that it is not necessary to sacrifice expressiveness in order to achieve automatic parallelisation and coordination of services. We believe that workflow languages stand to benefit greatly from following a more principled design approach which leverages well-established ideas from existing programming language theory, instead of ad-hoc solutions designed with a narrow perspective focusing on simple, coordination-only cases, as existing workflow languages have been.

8.3 Contributions

The key contributions made in this thesis are as follows:

1. **Illustration of lambda calculus as a workflow model (Chapter 3).** The basic computational model underlying our work is lambda calculus. It should of course be no surprise that this can be used to express workflows, since workflows are programs, and all programs can be expressed in lambda calculus, due to its Turing-complete nature. However, the abstract reduction model used for expression evaluation in lambda calculus does not address implementation concerns such as the mapping of functions to remote operations. By describing the relationships between lambda calculus and common features of workflow languages (Section 3.6), as well as defining a way for functions to interact with services over a network (Section 3.5.8), we have demonstrated how this model is suitable for data-oriented workflows. Our programming model, ELC, extends lambda calculus with a small collection of built-in functions and data types necessary for practical usage.

2. **An implementation of our programming model (Chapter 4).** We developed and gave a detailed description of the NReduce virtual machine, which executes programs written in ELC. This virtual machine is based on existing techniques for implementing functional programming languages, but adapts them in ways necessary to support workflow execution. In particular, it automatically detects parallelism in the absence of explicit annotations (Section 4.3), permits network I/O within the context of pure expressions (Section 4.5), and provides an efficient, purely functional list-based abstraction for arrays and data streams (Section 4.6). We addressed several performance-related issues that had not previously been explored in either the functional programming or workflow literature, in particular the management of large numbers of requests to avoid overloading servers (Section 6.2), and the effective load distribution of service requests (Section 6.4).

3. **An architecture for distributed workflow choreography (Section 4.7).** Most virtual machines and workflow engines run on a single machine, and can only communicate with services in a client/server fashion. This centralised control of service composition can require significant numbers of unnecessary data transfers,

as all output data produced by service operations must first be sent back to the client, and then forwarded on to subsequent operations that consume the data.

We developed a technique for distributed choreography of services in which result data is sent directly from producers to consumers, saving bandwidth. Importantly, all scheduling and data transfer is managed automatically by the system, requiring no explicit direction from the programmer. Our approach requires no modifications to existing services, as we implement all choreography logic in virtual machine nodes that are physically co-located with the services being used.

4. **A method for supporting multiple workflow languages with a single execution engine.** Our design philosophy primarily concerns a general method of program evaluation, rather than specific language features. By defining a suitably general language which makes little in the way of assumptions about usage (Section 3.5), we have demonstrated how it is possible to separate the concerns of high-level language features and low-level execution mechanisms. Parallel graph reduction and network communication are handled entirely within the virtual machine, enabling implementations of other languages such as XQuery to focus purely on the expression of language constructs in terms of the intermediate language, ELC.

   This approach opens up the possibility of using our virtual machine to execute a range of different workflow languages. Our decision to expose service access at the level of TCP connections also permits the use of arbitrary data formats. Although XQuery is tied to XML and web services, the virtual machine can easily support workflow languages that use alternative forms of structured data.

5. **Use of XQuery as a workflow language (Chapter 5).** The popularity of web services and the need to perform data manipulation within workflows led us to explore the idea of using XQuery for developing workflows. The language's native support for XML makes it ideal for accessing and manipulating data received from or sent to web services, and its powerful language features enable application logic to be implemented within workflows as well. The fact that XQuery is a pure, side effect free functional language also makes it a candidate for automatic parallelisation.

   Our XQuery compiler uses ELC as a compilation target, so that all parallel execution and network interaction can be managed by the NReduce virtual machine. This demonstrates the previous point of how higher-level workflow languages can be supported on top of our basic workflow definition and execution model.

6. **Performance evaluation (Chapter 7).** Our final contribution in this project has been an investigation into the performance gains possible using our workflow system. We have explored this both in the context of simple text-based services coordinated using workflows written directly in ELC (Sections 7.2 – 7.4), as well as XML-based web services coordinated using XQuery (Section 7.5). In both cases, significant speedups were demonstrated for the sample workflows, indicating that automatic parallelisation is practical in these scenarios, and that substantial reduction in bandwidth requirements can be achieved using our approach to choreography.

We also demonstrated that moderate levels of intermediate computation and data processing are feasible within workflows developed using our system. Performance-wise, our ELC implementation is competitive with widely-used scripting languages (Section 7.6.1), and is thus suitable for lightweight computation.

8.4 Future work

Although our main theoretical goals have been achieved, and we have produced a working prototype to demonstrate our ideas, there are several ways in which the work presented in this thesis could be taken forward. In the following sections we shall discuss both implementation improvements and additional research issues that we believe would be worthwhile exploring.

8.4.1 Compiler optimisations

Although the compute-intensive performance of NReduce is adequate for lightweight computation, it would be desirable to achieve faster execution speeds to enable more complex application logic to be implemented within a workflow. The effort that would be involved in doing so is largely a matter of software engineering rather than research, as it would primarily involve the application of existing techniques for compiler optimisation.

One of the main factors that would improve performance is to have the generated code make better use of registers. Currently, the code sequence generated for each bytecode instruction loads and saves the necessary variables from the current frame's memory area. This could be improved by caching values in registers, and only saving them back to the frame when a function call or return occurs.

Additional source-level optimisations could be applied prior to bytecode generation, to transform certain types of expressions into alternative forms which compute the same result, but more efficiently. In particular, *deforestation* [319] (also known as *fusion* [65]) could be applied to avoid many cases of temporary list creation, by merging the producer of a list with its consumer. Other techniques such as common sub-expression elimination, constant folding, and partial evaluation could also be applied.

The addition of static typing to ELC would also improve performance by avoiding the many dynamic type checks that are presently needed at runtime. This would not necessarily require the programmer to specify the types of functions and variables, since type inference [59, 235] could be used to determine this information automatically. Support for user-defined structured data types would also avoid the costs associated with representing objects as lists, which are a major factor in the performance of our XQuery implementation.

8.4.2 XQuery

As mentioned in Section 5.6.2, our XQuery prototype supports only a subset of the language. This is because were were focused on research issues rather than trying to produce a production-quality implementation. Now that we have demonstrated that our implementation approach is viable, it would be worthwhile adding support for all of XQuery, so that the system can be of more practical benefit.

Compute-intensive performance could be greatly improved by using statically-determined type information to skip many of the runtime checks and intermediate data structures that are presently used. As with ELC itself, all data objects in our XQuery implementation are tagged with type information, which is checked whenever a type-dependent operation such as addition is performed. Even higher costs are incurred for XQuery than for ELC in this respect, since individual values are always stored in sequences, which must be constructed and have their cardinality checked regularly during evaluation. Modifying the compiler to annotate each expression with type and cardinality information would permit sequence construction and deconstruction, as well as object type tagging and checking, to be skipped in certain situations where these operations can be determined to be redundant.

As a very high-level language, XQuery contains many language constructs which can often be inefficient if implemented naïvely. Most existing implementations of the language perform many source-level transformations on the code in order to avoid unnecessary work at runtime [177, 128]. For example, the simplest way to evaluate a path expression specifying that the first child of an element should be returned would be to apply the positional predicate to each item in the list, and return only those which match. In this particular case though, the predicate will only evaluate to true for the first item in the list, so it would be more efficient to simply return that item directly. This is one of the many transformations that would normally be included in an optimising XQuery compiler. These optimisations are largely a separate concern to code generation, so it would be possible to re-implement transformations performed by other XQuery processors in our own compiler.

Since our XQuery implementation is built on top of ELC, any improvements to the latter would also be of benefit. As mentioned in the previous section, inclusion of support for user-defined structured data types would permit us to represent objects such as XML tree nodes in a manner which would greatly reduce the costs of object construction and field access.

8.4.3 Fault tolerance

An important concern for all distributed applications is how to handle failures. Currently, our system aborts workflow execution as soon as an error occurs. Given the potentially long-running nature of workflows, and the possibility of network errors and machine crashes, it is highly desirable to handle these problems gracefully and enable workflow execution to continue wherever possible.

Our programming model assumes that all services are idempotent, which means that invoking a given service multiple times with the same inputs will always produce the same output. This makes it possible to simply retry a service operation if an error occurs. This would be useful for both temporary network failures, and faults occurring on individual nodes in a server farm in which a front-end load balancer can assign the retry request to a different, still-active machine.

Although conceptually simple, implementing this form of fault tolerance may be complicated in the case of lazy evaluation, since some computation may have already been performed on a portion of the response that has already been received from a service. In this case, the virtual machine may have to track which expressions are dependent upon on service results, or provide an exception handling mechanism to allow the retry logic to be expressed in user code.

Another aspect of fault tolerance is the failure of individual nodes in distributed deployments of NReduce used for choreography. Such failures would often result in the loss of a portion of the graph that is needed by expressions on other nodes. It may be possible to recover these by re-computing the lost data, provided the original expressions from which it was derived remained available. We anticipate this being substantially more complex to implement than the service retry mechanism.

8.5 Conclusions

Distributed computing is becoming widespread as business and scientific users increasingly make use of large numbers of computers working in parallel to carry out compute-intensive processing. Workflow languages have emerged as a means of coordinating this processing in an easy-to-use manner. While successful at exploiting parallelism of external services, existing workflow languages suffer from very limited programming models that do not support the expression of internal computation or data manipulation.

In this thesis, we have addressed the problem of developing a suitably expressive workflow language by adapting ideas from the functional programming community — in particular, the model of lambda calculus. This model is well-suited to expressing data-oriented computation of the sort used in many workflows, and is amenable to automatic parallelisation, due to its side effect free nature. As a Turing complete model of computation, lambda calculus can be used for arbitrary computation and data manipulation. With a small number of extensions in the form of built-in functions, data types, and syntactic sugar, lambda calculus provides a solid basis for a realistic programming language.

We have shown that by providing these extensions, as well as a purely data-oriented abstraction for interacting with services over a network, it is possible to use a lambda calculus-based language for defining workflows. This requires the application of appropriate implementation techniques to allow multiple services to be invoked in parallel. We have demonstrated how existing techniques for implementing parallel functional programming languages, such as parallel graph reduction, can be adapted to meet the specific requirements of workflows. In addition, we have shown how higher-level languages, such as

XQuery, can be supported on top of ELC, making our basic programming and execution model applicable to a wide range of languages.

We view workflow languages as a special class of programming language, which should be treated in much the same manner as other types of programming languages in terms of theory and implementation. The specific requirements of workflow languages — automatic parallelisation, integrated service access, and ease of use — should be addressed as language features to be implemented *in conjunction with* other standard features such as control flow constructs, primitive operations, and data types. There is no fundamental reason why workflow languages need to be any less powerful than general-purpose programming languages. We hope that this thesis will change the way that people think about workflow languages, and foster a greater degree of cooperation between the distributed computing and programming language communities in terms of integrating mutually beneficial ideas.

Appendix A

Lambda Calculus and Graph Reduction

Lambda calculus [67] is an abstract model of computation based on the concept of performing a series of transformations on an expression to reduce it to a simplified form. Invented in the 1930s, it is one of the earliest models of computation, and provides the underlying model upon which functional languages are based. It is Turing complete, which means that it can be used to express any program that can be executed by a computer.

A program in lambda calculus is an *expression*, which consists of *functions*, *applications*, and *symbols*. A symbol is just an identifier. A function consists of a single argument and a body, where the body itself is an expression. An application is a call to a function with a particular argument, and consists of a function followed by an expression.

An example of a simple expression is shown below. This consists of a function which takes an argument x, and whose body is a simple expression containing only x. This particular function is the *identity function*, because it simply returns the value that is passed to it. The outer expression is an application of this function to the symbol y.

$$(\lambda x.x)\ y$$

Evaluation of an expression consists of a sequence of *reductions*. Each reduction modifies the expression by replacing it with a simpler expression that has the same meaning. The evaluation of the expression above would proceed as shown below. The application is replaced with the body of the function, with all occurrences of x replaced with the argument y. In this particular case, only one reduction is necessary, since the resulting expression y cannot be simplified any further. An expression for which no further reductions can be made is said to be in *normal form*.

$$\begin{aligned} & (\lambda x.x)\ y \\ \rightarrow\ & y \end{aligned}$$

Functions of multiple arguments are defined by nesting several single-argument functions. When a call is made, the arguments are substituted one-by-one. The first reduction is equivalent to calling the outer definition with a single argument, which will return a new function that accepts the remaining arguments. This proceeds until all arguments have been processed. Functions defined in this manner are called *curried functions*.

The following example contains a curried function with two arguments, x and y. The first step in evaluation is to substitute for x, which leaves a modified version of the expression containing a new function of one argument, y. This function is then applied to the next argument, which in this case is the identity function. Finally, the same step as in the previous example is used to obtain the final result, which is v.

$$\begin{aligned} & (\lambda x.\lambda y.y\,x)\ v\ (\lambda z.z) \\ \to\ & (\lambda y.y\,v)\ (\lambda z.z) \\ \to\ & (\lambda z.z)\ v \\ \to\ & v \end{aligned}$$

A.1 Reduction order

There are two different ways to reduce an expression. One is *applicative order*, in which the argument to a function application is reduced prior to the application itself. The other is *normal order*, in which the application is reduced first. In the latter case, the argument will only be reduced if and when it is actually needed. If it is not used within the function body, then it will never be reduced; otherwise, its reduction may occur at some later point in time.

The following steps show how the same expression is reduced in applicative and normal order:

$(\lambda x.\lambda y.y)\ ((\lambda z.z)\ w)\ v$	$(\lambda x.\lambda y.y)\ ((\lambda z.z)\ w)\ v$
$\to\ (\lambda x.\lambda y.y)\ w\ v$	$\to\ (\lambda y.y)\ v$
$\to\ (\lambda y.y)\ v$	$\to\ v$
$\to\ v$	
Applicative order	Normal order
(strict evaluation)	(lazy evaluation)

With applicative order reduction, the expression $((\lambda z.z)\,w)$ is first reduced. Then, the application of the λx function to the argument is reduced, which ends up discarding the w because the symbol x is not referenced from within the function body. With normal order reduction, the function application happens first, and the unnecessary reduction to w is avoided.

Applicative order reduction is often referred to as *strict evaluation*, because it requires all parts of an expression to be evaluated. This is always the case for imperative languages, which evaluate of the the arguments supplied to a function before the actual function call occurs. Normal order reduction is known as *lazy evaluation*, because it only evaluates expressions if and when they are needed. Lazy evaluation can potentially be faster than strict evaluation, since it may avoid doing unnecessary work. However, this depends very much on the program, and it carries certain implementation costs.

Lazy evaluation is very useful for implementing many higher-level constructs in lambda calculus, such as conditionals and recursion. For example, consider the following definition of the *if* function, and the corresponding representations of true and false:

$$\begin{aligned} \text{if} &= \lambda c.\lambda t.\lambda f.c\ t\ f \\ \text{true} &= \lambda t.\lambda f.t \\ \text{false} &= \lambda t.\lambda f.f \end{aligned}$$

The *if* function takes three arguments: a conditional expression, a true branch, and a false branch. It uses a representation of boolean values as functions of two arguments, where *true* returns its first argument, and *false* returns the second. *if* works by applying one of these functions to the true and false branches, so that if the conditional is true, the result of the expression will be the true branch, otherwise it will be the false branch:

$$\begin{aligned} & \text{if true } a\ b \\ \to\ & (\lambda t.\lambda f.t)\ \ a\ b \\ \to\ & (\lambda f.a)\ \ b \\ \to\ & a \end{aligned}$$

Now consider a case where b corresponds to some complex expression that would take a long time to compute. We don't want to evaluate it unnecessarily, and since the conditional here is true we know its value is unneeded. In the following example, evaluating the third parameter to *if* would actually cause the program to go into an infinite loop, since the expression $(\lambda x.x\ x)\ (\lambda x.x\ x)$ simply evaluates to itself. By using lazy evaluation, we can avoid having to compute the value altogether:

$$\begin{aligned} & \text{if true } a\ \ ((\lambda x.x\ x)\ \ (\lambda x.x\ x)) \\ \to\ & (\lambda t.\lambda f.t)\ \ a\ \ ((\lambda x.x\ x)\ \ (\lambda x.x\ x)) \\ \to\ & (\lambda f.a)\ \ ((\lambda x.x\ x)\ \ (\lambda x.x\ x)) \\ \to\ & a \end{aligned}$$

Lazy evaluation is necessary to implement recursive functions in lambda calculus, since any recursive function must somewhere contain a conditional which tests for when to stop.

A.2 Additional functionality

Lambda calculus does not directly provide support for numbers, conditional expressions, or data structures, but it is possible to implement these in the language itself. For example, one way of representing a number is in terms of a zero function and a successor function; a number is then a function which accepts these two values as arguments, and applies the successor function the appropriate number of times to zero. Operations like addition can be defined in terms of these representations, though we will not go into such detail here. As an example, the number 3 could be defined as shown below:

$$3 = 1 + 1 + 1 + 0 = \lambda s.\lambda z.s\,(s\,(s\,(z)))$$

Data structures can be implemented using linked lists, and linked lists can be implemented as pairs of values, where the first value is the data item, and the second value is a link to the next pair. A pair can be represented as a function of three arguments, where the first two arguments correspond to the values in the pair, and the third is a *selector function* that is applied to the first two arguments. Depending on the selector function supplied, it could extract either the first or the second value.

The function used to create a pair is called *cons*. When it is used within an expression, it is only called with two arguments, even though it actually requires three. It is thus a partial application, which can only be evaluated later on when the third argument is supplied. Subsequently, either *head* or *tail* can be called with the pair as a parameter. They supply a third argument to the pair function which enables either the first or second argument of the pair to be extracted:

$$\begin{aligned} \text{cons} &= \lambda h.\lambda t.\lambda s.s\ h\ t \\ \text{head} &= \lambda p.p\ (\lambda h.\lambda t.h) \\ \text{tail} &= \lambda p.p\ (\lambda h.\lambda t.t) \end{aligned}$$

Linked lists can be constructed by repeated calls to *cons*, and elements can be extracted using the *head* and *tail* functions. The following example creates a list of three elements, with the tail represented by the (undefined) symbol *nil*, and then extracts the second element:

$$\begin{aligned} & \text{head}\ (\text{tail}\ (\text{cons}\ a\ (\text{cons}\ b\ (\text{cons}\ c\ \text{nil})))) \\ \rightarrow\ & \text{head}\ (\text{cons}\ b\ (\text{cons}\ c\ \text{nil})) \\ \rightarrow\ & b \end{aligned}$$

This is a very powerful mechanism, because it enables a wide variety of data structures to be implemented. For example, a string can be represented as a list of numbers, each of which is a character code. An XML element can be represented as a list containing strings for the namespace and local name, as well as nested lists for the attributes and child elements. In fact, literally any data structure can be built on top of these basic mechanisms.

It is this power through simplicity that makes lambda calculus so attractive as a model of computation. When writing a compiler or interpreter for the language, it is only necessary to provide explicit support for a few very basic features. Programmers can then define their own higher-level abstractions on top of these to create arbitrarily complex programs. Of course, efficiency is a concern; it is very simple to write an interpreter for lambda calculus, but running programs at a speed competitive with typical interpreted languages takes a lot more work. Chapter 4 discusses the approach we have taken to doing this.

A.3 Extensions

Although numbers, conditionals, and logical operators can be implemented within lambda calculus itself, doing so is awkward. Such implementations are also very inefficient to execute. Since computers are already very capable of doing these things in hardware, it makes sense to expose these capabilities to the language. Doing so also provides significant opportunities for optimisation within the compiler.

Virtually all programming languages based on lambda calculus provide native support for numbers, arithmetic functions, logical operators, conditional statements, and *cons* pairs. These can be implemented as built-in functions and value types, and do not require extensions to the syntax or the semantics described above; their implementation is thus more akin to a standard library than extensions to the core language itself. The relationship between these constructs and lambda calculus is analogous to that between the standard Java APIs and the core Java language. The language definition stands on its own, and the APIs are separately defined functionality that can be used from within the language, but do not extend or change its semantics.

Sophisticated functional languages provide many syntactic extensions, such as pattern matching, list comprehensions, and infix operators. All of these have equivalent representations in terms of the elementary constructs of lambda calculus, and the extended syntax is generally transformed into a variant of lambda calculus as one of the early compilation steps.

The point at which a language can no longer be considered lambda calculus and should instead be referred to by a different name is not necessarily clear. We use the name ELC (*extended lambda calculus*) in this thesis to refer to our variant of the language, since we have deliberately kept the number of extensions to a minimum. From a theoretical perspective, most of the conceptual issues explored in this project are not specific to our particular extensions. Many of the ideas we have investigated could thus be applied to other functional languages as well.

A.4 Graph reduction

The most common way of evaluating lambda calculus expressions is using *graph reduction*. Initially, the graph corresponds to the syntax tree of the expression, with each symbol as a leaf node, and function definitions and applications as branch nodes.

Reduction proceeds by traversing the graph to find the first *redex*, which is an application node eligible for reduction. For normal order reduction, this is done by traversing the graph along the left edges of application nodes until a function is encountered. For a function of one argument, the redex will be the application node directly above it. The redex is replaced with a copy of the function body in which the supplied value is substituted for all instances of the argument name.

An implementation can provide built-in functions, which may take multiple arguments. The redex is the application node corresponding to the application of that function to

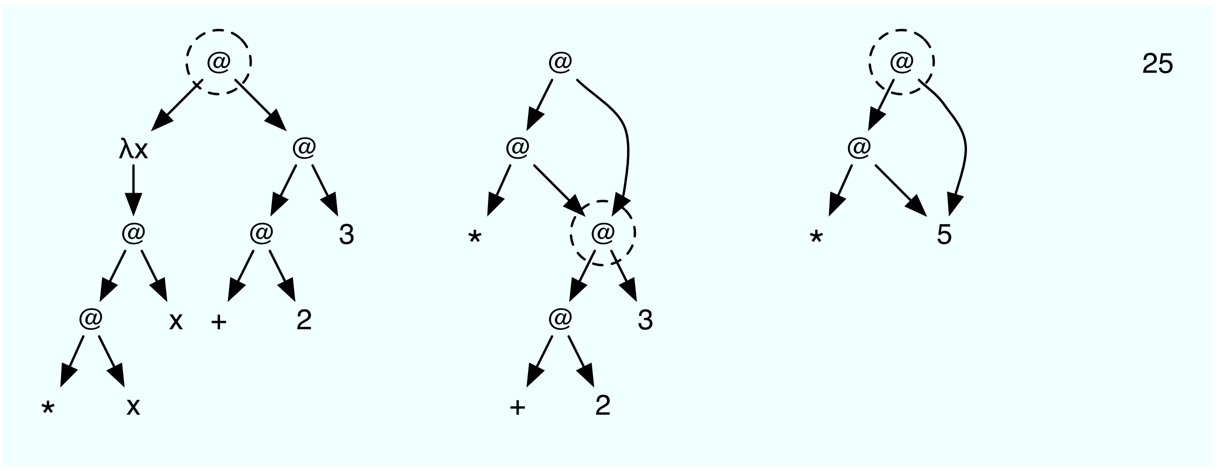


Figure A.1: Graph reduction

the correct number of arguments. Reduction in this case consists of invoking the built-in function with the relevant values, and replacing the redex with the result.

Figure A.1 depicts the graph reduction steps for the evaluation sequence shown below; in each step, the current redex is circled. In the first step, the function $(\lambda x. * \; x \; x)$ is instantiated. Next, the built-in $+$ operation is invoked, and subsequently * is applied to the result. The final version of the graph contains only one value, the number 25.

$$\begin{array}{ll} & (\lambda x. * \; x \; x) \;\; (+ \; 2 \; 3) \\ \rightarrow & * \; z \; z \quad \text{where } z = (+ \; 2 \; 3) \\ \rightarrow & * \; 5 \; 5 \\ \rightarrow & 25 \end{array}$$

This example shows the benefits of performing reduction using a graph rather than a tree. The function defined in this expression uses its parameter multiple times. With a tree, instantiating this function would require two separate copies of the expression (+ 2 3), each of which would have to be reduced independently. Because a graph is used, each occurrence of the symbol x can be replaced with a reference to the single instance of the actual parameter. This is referred to as *sharing*.

A.5 Parallel evaluation

Sometimes, there is more than one sequence of reductions that can be performed on an expression. For example, $* \; (+ \; 2 \; 4) \;\; (+ \; 3 \; 4)$ can be reduced in the following two ways:

$$\begin{array}{llcll} & * \; (+ \; 2 \; 4) \;\; (+ \; 3 \; 4) & \qquad & & * \; (+ \; 2 \; 4) \;\; (+ \; 3 \; 4) \\ \rightarrow & * \; 6 \;\; (+ \; 3 \; 4) & & \rightarrow & * \; (+ \; 2 \; 4) \;\; 7 \\ \rightarrow & * \; 6 \; 7 & & \rightarrow & * \; 6 \; 7 \\ \rightarrow & 13 & & \rightarrow & 13 \end{array}$$

Both of these reduction sequences produce the same result, the number 13. One might wonder if this is true for all expressions; is it possible for two different reduction sequences to produce different values? Fortunately, the answer is no, and this is due to the Church-Rosser theorem [69], which states two important properties:

1. For two expressions A and B that are convertible, there exists an expression E such that A can be reduced to E and B can be reduced to E
2. For two expressions A and B where B is in normal form and A is reducible to B, normal order reduction will produce B from A.

Property 1 has the important implication that it is not possible to produce a different result through different reduction sequences from a given expression. By *convertible*, it means that both expressions have the same meaning, and can be derived from each other or a common source. Thus, A and B can both be partially reduced versions of an original expression, such as the second step in the example above, and this property states that they cannot produce different values.

Property 2 says that if it is possible to produce a result from a given expression, then normal order reduction will give that result. As we have seen in Section A.1, it is possible for applicative order to cause non-termination of a program, by attempting to evaluate expressions that cause infinite loops but whose value is not actually needed by the program. Normal order evaluation will always produce a normal form, unless the program itself is non-terminating. Thus, it is safe to use, since we know we can get the correct result for any program which terminates.

The Church-Rosser theorem has profound implications for the way in which expressions can be evaluated. Because it is not possible to produce different results by using different reduction orders, we can safely perform multiple reductions in parallel. Consider the alternative reduction sequence for the example above, where we reduce both (+ 2 4) and (+ 3 4) at the same time:

$$\begin{aligned} & * \ (+\ 2\ 4)\ \ (+\ 3\ 4) \\ \rightarrow\ & * \ 6\ 7 \\ \rightarrow\ & 13 \end{aligned}$$

This is not possible with imperative languages, because functions can have side effects. A function call used within an expression could modify a global variable which is used by another function in a later part of the expression, and if the two were evaluated in a different order then a different result could be produced. In pure functional languages, all functions and expressions are side effect free, and thus it is safe to perform evaluation in any valid order.

The way in which this parallelism occurs is an issue for the runtime system; several approaches are possible, including evaluating expressions on different processors within the same computer, or performing evaluation using multiple computers. The latter case requires mechanisms for distribution of the expression graph and communication between different machines. These implementation issues are discussed further in Section 2.4.5.

Appendix B

XQuery Compilation

In this appendix, we give a more detailed description of our XQuery compiler than was included in Section 5.5.6. While a full coverage of all implementation details is outside the scope of this thesis, our aim here is to illustrate the main principles used in the compiler, and to convey an understanding of the way in which it operates.

The main purpose of this appendix is to illustrate how XQuery can be translated into lambda calculus. This is significant because it demonstrates that all of the complexity of XQuery can be completely separated from the underlying evaluation mechanism. This separation of concerns contributed greatly to the success of our implementation. Our XQuery compiler and support libraries contain no explicit knowledge of parallelism; simply by expressing all computation in terms of lambda calculus, they open up the possibility of exploiting the parallelism inherent in this computation.

B.1 Overview

The compilation process consists of several stages. First, the source code is parsed into an abstract syntax tree in the traditional manner. Next, a search is performed for `import service` statements; for each of these, the WSDL file is downloaded and a series of stub functions are generated, corresponding to the operations provided by the service. A simplification process is then applied to the XQuery code to resolve all namespace prefixes and convert certain language constructs into a core set that reduces the number of cases that need to be considered during code generation. After this, the syntax tree is traversed to translate each XQuery language construct into an equivalent ELC expression, many of which make calls to functions provided by the runtime library. Simple optimisations are then performed on the generated code, after which top-level namespace declarations and a `main` function are added to the code. The following sections describe these steps in more detail.

```
foreach3 lst f =
(letrec
  length = (len lst)
  loop   = (\cur.\pos.
             (if cur
               (++ (f (head cur) pos length)
                   (loop (tail cur) (+ pos 1)))
               nil))
in
  (loop lst 1))
```

Figure B.1: The `foreach3` function, contained in the runtime library

B.2 Format of compilation rules

The compiler itself is implemented in *Stratego* [49], a language designed for program transformation. It provides a high-level syntax consisting of a list of rules, each of which has a pattern and an action. Various traversal strategies may be applied to a syntax tree, each of which applies the rules in the order of the traversal. A rule whose pattern matches a given node in the tree causes the action to be evaluated; the node is then replaced with the result of the action. A set of rules defining the translation from one language to another thus appears similar to denotational semantics [265], particularly in the case where the target language is (a variant of) lambda calculus. Although the full details of the translation rules used by our compiler are outside the scope of this thesis, the following excerpt from the source code of the compiler gives an idea of what these translation rules look like:

```
E : |[ e1/e2  ]|   ->   |{ (xq::foreach3 e1 (\citem.\cpos.\csize.e2)) }|

E : |[ e1[e2] ]|   ->   |{ (xq::filter3 (\citem.\cpos.\csize.
                                            (xq::predicate_match cpos e2))
                                          e1) }|
```

The first rule above defines the translation performed for path step expressions. The expression `e1` produces a sequence of items which is to be iterated over, at each step evaluating the expression `e2` in a modified dynamic context in which the current item, position, and size are set appropriately for a given position in the sequence. This iteration is performed by the `foreach3` library function, shown in Figure B.1. The function is similar to `map`, except that it provides the position and size of the list to the supplied function, and concatenates the results of each call (to preserve XQuery's requirement that sequences cannot be nested). The second rule defines the translation for filter expressions. These are processed similarly to path expressions, except that the `filter3` function is used instead. This function, whose implementation is similar to that of `foreach3`, traverses the list and returns all of the items for which the supplied function returns true.

Note the use of lambda abstractions within both of these rules. These define anonymous in-line functions which correspond to a loop body, which are passed to higher-order functions that implement the iteration logic. Also note the way in which the dynamic context is represented — as discussed in Section 5.5.4, we rely on the scoping rules of variables to define a new dynamic context within the scope of a particular expression. It is possible for multiple loop iterations to be under evaluation at the same time, in which case each iteration will have a separate dynamic context. Each of these can coexist without interfering with others, since they use separate instantiations of the lambda abstractions with different values bound to the `citem`, `cpos`, and `csize` parameters.

In addition to rule definitions of the form described above, Stratego also provides other language constructs which can be used to implement the parts of a compiler which do not logically correspond to the pattern/action model. Stratego is actually a functional language, and the expression syntax it provides for non-rule-based parts of the program is similar to that of other functional languages. We use these features of the language to implement certain parts of the compiler, such as WSDL processing and namespace resolution. Overall, the use of Stratego substantially simplified the task of writing the compiler in comparison to more traditional approaches involving imperative languages.

In hindsight, Stratego would have been a superior choice to C for implementing the bytecode compiler in NReduce as well, although we were only introduced to Stratego later in the project, after NReduce was largely completed.

B.3 Initial compilation stages

The first stage after parsing is to scan through the syntax tree for `import service` statements and generate the stub functions described in Section 5.3.3. For each `import service` statement encountered, the WSDL file at the specified address is downloaded and parsed. The compiler analyses the file to determine what operations are provided by the service, their parameter names and types, and the binding information necessary to invoke the operations. This information is used to generate the stub functions, which build a SOAP envelope with the appropriate element names and a body including the supplied parameters. Once this stage is complete, the stub functions appear to the rest of the compiler exactly like normal user-defined XQuery functions. An example of a generated stub function was shown in Figure 5.4 on page 164.

The next step is to resolve namespace references used within the code. XML and XQuery use *prefixes* to refer to namespace URIs. Bindings between prefixes and URIs must be established either by declarations given at the top of the file, or within literal element constructors. A complication that arises with the latter form is that it is possible for a given prefix to be bound to different namespace URIs in different parts of the file, and vice-versa. This means that the compiler cannot rely on a 1:1 mapping between the two. To alleviate this complexity, all namespace-qualified function and variable names are transformed into unqualified names, with the bindings present at each point in the syntax tree used to resolve references to their targets. The unqualified names incorporate a unique, automatically-generated identifier, to prevent clashes with other names. As a result of this process, all functions and variables are referenced by unique identifiers without namespace prefixes, simplifying the remaining stages of the compiler.

Some XQuery language constructs, such as element constructors and `for` loops, have several different possible forms. To minimise the number of cases that need to be considered during code generation, certain forms of expressions are translated into alternative but semantically identical forms. Most of these are simply abbreviations which can be expanded — for example, an element constructor with no expression specifying its children is transformed into an element constructor with a child expression corresponding to the empty sequence. Abbreviated path steps are handled in a similar manner — for example, `//` is translated into `/descendant-or-self::node()/`.

B.4 Code generation

The main part of the compiler is the *code generation* phase, in which the syntax tree is transformed from (simplified) XQuery code into ELC. It is this stage which carries out the bulk of the compilation logic, and dictates how the various language constructs provided by XQuery can be expressed in terms of ELC. In this section, we describe how each of the main language constructs are translated.

- **Global namespace declarations** are transformed into top-level variable definitions of the form `ns_P = "U"`, where `P` is the prefix, and `U` is the URI. Any reference to a namespace prefix from within the code corresponds to one of these definitions, so that the URI associated with a particular prefix may easily be obtained, for example when constructing an element. Local namespace declarations appearing within literal elements cause `letrec` bindings of the same form to be generated, which override the global definitions within the scope of a particular portion of the syntax tree.

- **Arithmetic expressions** are handled by library functions, which take care of the special semantics of the built-in XQuery operators, which differ from those of ELC. Before performing an arithmetic operation, XQuery requires that the operands are atomised (converted from nodes to atomic values, if necessary). The operands must also be checked to ensure that they are sequences of exactly one item, and that the atomised values are either untyped or are of a valid numeric type. It is only after these conversions and checks have been performed that the actual operation can be performed, and afterwards, the result must be wrapped up in a sequence again. This logic is implemented within a library function, which the compiler generates a call to.

- **Comparison operators** are handled similarly to arithmetic operators. For *value comparisons*, which are defined to compare sequences of exactly one element each, the operands are passed to a library function which performs the atomisation and then the actual comparison. For *general comparisons*, which allow sequences of multiple items, and compare every item in the first sequence against every item in the second, another set of library function is used to iterate over the sequences.

- **If statements** are translated into calls to ELC's built-in `if` function, with the true and false branches passed in directly, and the conditional test first passed to a library function which computes its effective boolean value.

- **Element constructors** are compiled into a call to a library function which is passed the name, attributes, and children of the element as parameters. The attributes and children are obtained by evaluating the body of the constructor, and splitting the resulting sequence into two parts — one containing the attributes, and the other containing all other nodes. If a qualified name is used, the prefix must be resolved to a namespace URI. For element names given explicitly in the code, this resolution occurs statically, using the previously-generated variable associated with the specified prefix. For dynamically-computed element names, the resolution occurs at runtime using the in-scope `namespaces` variable, as discussed in Section 5.5.3. Attribute, text, and other types of constructors are compiled in a similar manner to element constructors.

- **Path steps** of the form E_1/E_2 cause the expression E_2 to be evaluated for every node in the sequence returned by E_1. This is similar to the idea of using `map` to apply a lambda abstraction to every value in a list, except that in this case the results of the function calls must be concatenated instead of being added to another list, since XQuery does not permit nested sequences. The function must also take three arguments instead of one: the context item, context position, and context size. As discussed in Section B.2, the `foreach3` function (a variant of `map`) is used for this purpose; the function is passed a lambda abstraction whose body is E_2.

- **Node tests**, which are used within path steps, act like filters which select only those nodes in a sequence which have a specified name or type. Node tests are always associated with an axis (which defaults to `child`), and the node test/axis pairing is compiled as a single unit into a call to the built-in `filter` function. The function parameter to `filter` is one of the library functions which selects the appropriate node, which in the case of node names is a partial application of the name test function to the specified name. The list parameter to `filter` is the sequence of nodes obtained from searching the corresponding axis of the context node, such as `child`, `parent`, `descendant`, or `attribute`. These searches are performed by library functions, defined for each possible axis.

- **Filter expressions** of the form $E_1\,[E_2]$ cause E_2 to be evaluated for every value in the sequence E_1, resulting in only those values of E_1 for which E_2 evaluates to `true` being returned. This is analogous to the `filter` function supported by most functional languages. The only difference is that as with path steps, the lambda abstraction of which E_2 forms the body must receive the context item, position, and size, instead of just a single argument. As mentioned in Section B.2, a variation of ELC's built-in `filter` function is used for this purpose.

- **For loops** operate similarly to path steps, in that a loop body is evaluated once for every item in the sequence being iterated over, with the results concatenated together. With `for` loops however, the programmer specifies a variable name which refers to the current item. The loop body is translated into a lambda abstraction which takes the current item as a parameter, whose name is the specified variable. This lambda abstraction is passed to a `map`-like function which concatenates the results. Another variation of `map` is used if the `for` expression contains an `at` clause, which specifies an additional variable which is assigned the current position for each iteration. In this case, the body's lambda abstraction has two parameters.

- **Let bindings** are equivalent to `letrec` expressions in ELC, and are translated directly.

- **Function calls**, which in XQuery may take multiple parameters, are translated into nested sequences of single-argument function applications, which is how they are represented in ELC syntax trees. An ELC expression of the form `(f e1 e2 e3)` is parsed as `(((f e1) e2) e3)`; the former notation is simply a convenient shorthand used when writing code. The code generated for an XQuery function call of $n > 1$ arguments is therefore translated into a function call of $n - 1$ arguments, with the result of this call applied to the last argument. This transformation is applied repeatedly until each function application node in the syntax tree has only a single argument.

B.5 Final compilation stages

Once code generation is complete, a set of simple optimisations are performed on the generated ELC code. These optimisations are divided into two categories: those that are specific to the XQuery compiler, and those that are generally useful for all ELC code. Only a small number of XQuery-specific optimisations are applied, mainly addressing cases in which certain operations can be skipped because it is known that a value is already of a particular type, such as when a boolean value is constructed and then its effective boolean value is immediately computed. The ELC optimisations primarily deal with avoiding redundant list operations. The latter could theoretically be implemented directly in NReduce, but are simpler to express in the XQuery compiler as it is implemented in Stratego rather than C. Since our prototype implementation is intended purely as a means of demonstrating our ideas, we have not gone to the effort of implementing more advanced optimisations like those performed by production-quality implementations of XQuery [127, 89]. The optimisations our compiler makes only address a very small set of obvious cases, though these could be expanded on in future work.

After optimisation, a couple of final additions are made to the generated code. Top-level namespace declarations from the XQuery source are collated together and placed in a global `namespaces` variable which is used for namespace resolution during construction of elements with names that are computed at runtime, as explained in Section 5.5.3. This variable is in scope within all parts of the generated code, though it may be overridden in parts of the query when additional namespaces are defined. Finally, a `main` function is added, which acts as the entry point for the program, and allows an input XML document to be optionally read from a file, specified as a command-line parameter. The `main` function runs the query, and then returns the resulting XML tree serialised into textual format, which is printed to standard output.

The output of the compiler is an ELC source file which may then be executed by NReduce. To simplify operation, a wrapper script is used to invoke the compiler and then run the resulting output file. The user can thus compile and run the XQuery program using a single command.

Bibliography

[1] Jon Ferraiolo abd Jun Fujisawa abd Dean Jackson. Scalable Vector Graphics (SVG) 1.1 Specification. W3C recommendation, World Wide Web Consortium, January 2003.

[2] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, July 1996. Second edition.

[3] Serge Abiteboul, Angela Bonifati, Gregory Cobena, Ioana Manolescu, and Tova Milo. Dynamic XML documents with distribution and replication. In *Proceedings of the ACM SIGMOD Conference*, 2003.

[4] David Abramson, Colin Enticott, and Ilkay Altinas. Nimrod/K: towards massively parallel dynamic grid workflows. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[5] David Abramson, Jon Giddy, and Lew Kotler. High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid? In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 520–528, Cancun, Mexico, May 2000.

[6] Peter Achten and Marinus J. Plasmeijer. The Ins and Outs of Clean I/O. *Journal of Functional Programming*, 5(1):81–110, 1995.

[7] Loredana Afanasiev and Maarten Marx. An analysis of XQuery benchmarks. *Information Systems*, 33(2):155–181, 2008.

[8] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[9] R.J. Allan and M. Ashworth. A Survey of Distributed Computing, Computational Grid, Meta-Computing and Network Information Tools. Technical report, UKHEC, 2001.

[10] Ilkay Altintas, Oscar Barney, and Efrat Jaeger-Frank. Provenance Collection Support in the Kepler Scientific Workflow System. In Luc Moreau and Ian Foster, editors, *Provenance and Annotation of Data*, volume 4145 of *Lecture Notes in Computer Science*, pages 118–132. Springer Berlin / Heidelberg, 2006.

[11] Allen L. Ambler and Margaret M. Burnett. Visual forms of iteration that preserve single assignment. *Journal of Visual Languages & Computing*, 1(2):159–181, June 1990.

[12] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, New York, NY, USA, 1967. ACM.

[13] A. L. Ananda, B. H. Tay, and E. K. Koh. A survey of asynchronous remote procedure calls. *ACM SIGOPS Operating Systems Review*, 26(2):92–109, 1992.

[14] David P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *5th IEEE/ACM International Workshop on Grid Computing*, Pittsburgh, USA, November 2004.

[15] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[16] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business Process Execution Language for Web Services Version 1.1. http://ifr.sap.com/bpel4ws/, May 2003.

[17] Andrew W. Appel and David B. MacQueen. A Standard ML compiler. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 301–324, London, UK, 1987. Springer-Verlag.

[18] Apple Computer. Xgrid Guide. http://www.apple.com/acg/xgrid/, March 2004. Preliminary.

[19] Assaf Arkin. Business Process Modeling Language, November 2002.

[20] Assaf Arkin, Sid Askary, Scott Fordin, Wolfgang Jekeli, Kohsuke Kawaguchi, David Orchard, Stefano Pogliani, Karsten Riemer, Susan Struble, Pal Takacsi-Nagy, Ivana Trickovic, and Sinisa Zimek. Web Service Choreography Interface (WSCI) 1.0. W3C note, World Wide Web Consortium, August 2002. http://www.w3.org/TR/wsci.

[21] Joe Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, July 2007.

[22] K. Arvind and Rishiyur S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Trans. Comput.*, 39(3):300–318, 1990.

[23] Lennart Augustsson. A compiler for lazy ML. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 218–227, New York, NY, USA, 1984. ACM Press.

[24] Lennart Augustsson and Thomas Johnsson. Parallel graph reduction with the (v , G)-machine. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 202–213, New York, NY, USA, 1989. ACM Press.

[25] Bartosz Balis, Marian Bubak, and Bartlomiej Labno. Monitoring of Grid scientific workflows. *Scientific Programming*, 16(2-3):205–216, 2008.

[26] H.P. Barendregt, M.C.J.D. van Eekelen, P.H. Hartel, L.O. Hertzberger, M.J. Plasmeijer, and W.G. Vree. The Dutch Parallel Reduction Machine Project. *Future Generation Computer Systems*, 3(4):261–270, December 1987.

[27] R. Barga, J. Jackson, N. Araujo, D. Guo, N. Gautam, and Y. Simmhan. The Trident Scientific Workflow Workbench. In *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, pages 317–318, December 2008.

[28] Roger Barga and Dennis Gannon. Scientific versus Business Workflows. In I.J. Taylor, E. Deelman, D. Gannon, and M. Shields, editors, *Workflows for eScience - Scientific Workflows for Grids*, pages 9–16. Springer, December 2006.

[29] Adam Barker, Paolo Besana, David Robertson, and Jon B. Weissman. The benefits of service choreography for data-intensive computing. In *CLADE '09: Proceedings of the 7th international workshop on Challenges of large applications in distributed environments*, pages 1–10, New York, NY, USA, 2009. ACM.

[30] Adam Barker and Jano van Hemert. Scientific Workflow: A Survey and Research Directions. In *Proceedings of the The Third Grid Applications and Middleware Workshop (GAMW'2007), in conjunction with the Seventh International Conference on Parallel Processing and Applied Mathematics (PPAM'07)*, Gdansk, Poland, September 2007.

[31] Adam Barker, Jon Weissman, and Jano van Hemert. Orchestrating Data-Centric Workflows. In *The 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2008)*. IEEE, May 2008.

[32] Adam Barker, Jon B. Weissman, and Jano van Hemert. Eliminating the middleman: peer-to-peer dataflow. In *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*, pages 55–64, New York, NY, USA, 2008. ACM.

[33] Alistair Barros, Marlon Dumas, and Phillipa Oaks. Standards for Web Service Choreography and Orchestration: Status and Perspectives. In *Business Process Management Workshops*, volume 3812 of *Lecture Notes in Computer Science*, pages 61–74. Springer Berlin / Heidelberg, 2006.

[34] Khalid Belhajjame, Suzanne M. Embury, and Norman W. Paton. On Characterising and Identifying Mismatches in Scientific Workflows. In *Data Integration in the*

Life Sciences, volume 4075 of *Lecture Notes in Computer Science*, pages 240–247. Springer Berlin / Heidelberg, 2006.

[35] Nick Benton. *Strictness Analysis of Lazy Functional Programs.* PhD thesis, University of Cambridge Computer Laboratory, December 1992.

[36] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernandez, Michael Kay, Jonathan Robie, and Jerome Simeon. XML Path Language (XPath) 2.0. W3C recommendation, World Wide Web Consortium, January 2007.

[37] Shishir Bharathi, Ann Chervenak, Ewa Deelman, Gaurang Mehta, Mei-Hui Su, and Karan Vahi. Characterization of Scientific Workflows. In *The 3rd Workshop on Workflows in Support of Large-Scale Science (WORKS08), in conjunction with Supercomputing (SC08) Conference*, Austin, Texas, November 2008.

[38] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. Optimized Software Synthesis for Synchronous Dataflow. In *Intl. Conf. on Application-specific Systems, Architectures & Processors*, July 1997. invited paper.

[39] Gavin Bierman, Erik Meijer, and Wolfram Schulte. The essence of data access in Cw. In Andrew Black, editor, *ECOOP 2005 - Object-Oriented Programming, 19th European Conference*, volume 3586 of *Lecture Notes in Computer Science*, pages 287–311, Glasgow, UK, July 2005. Springer.

[40] Guy E. Blelloch, Jonathan C. Hardwick, Siddhartha Chatterjee, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. In *PPOPP '93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 102–111, New York, NY, USA, 1993. ACM.

[41] Adrienne Bloss, Paul Hudak, and Jonathan Young. Code Optimizations for Lazy Evaluation. *Higher-Order and Symbolic Computation*, 1(1):147–164, June 1988.

[42] Michael Blow, Yaron Goland, Matthias Kloppmann, Frank Leymann, Gerhard Pfau, Dieter Roller, and Michael Rowley. BPELJ: BPEL for Java, March 2004. http://ftpna2.bea.com/pub/downloads/ws-bpelj.pdf.

[43] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, and Jerome Simeon. XQuery 1.0: An XML Query Language. W3C recommendation, World Wide Web Consortium, January 2007.

[44] Andre Rauber Du Bois and Antonio Carlos da Rocha Costa. Distributed Execution of Functional Programs Using the JVM. In *Computer Aided Systems Theory - EUROCAST 2001-Revised Papers*, pages 570–582, London, UK, 2001. Springer-Verlag.

[45] Per Bothner. Compiling XQuery to Java bytecodes. In *XIME-P*, Paris, June 2004.

[46] Shawn Bowers and Bertram Ludascher. An Ontology-Driven Framework for Data Transformation in Scientific Workflows. In *Data Integration in the Life Sciences*, volume 2994 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin / Heidelberg, 2004.

[47] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple Object Access Protocol (SOAP) 1.1. W3C note, World Wide Web Consortium, May 2000.

[48] Don Box and Chris Sells. *Essential .NET, Volume I: The Common Language Runtime (Microsoft .NET Development Series)*. Addison-Wesley Professional, November 2002.

[49] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Science of Computer Programming*, 72(1-2):52–70, June 2008. Special issue on experimental software and toolkits.

[50] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and FranÃ§ois Yergeau. Extensible Markup Language (XML) 1.0 (Fourth Edition). W3C recommendation, World Wide Web Consortium, August 2006.

[51] Silvia Breitinger, Rita Loogen, Yolanda Ortega-Mallen, and Ricardo Pena-Mari. Eden - The Paradise of Functional Concurrent Programming. In *Euro-Par '96: Proceedings of the Second International Euro-Par Conference on Parallel Processing*, pages 710–713, London, UK, 1996. Springer-Verlag.

[52] Peter Brunner, Hong-Linh Truong, and Thomas Fahringer. Performance Monitoring and Visualization of Grid Scientific Workflows in ASKALON. In Michael Gerndt and Dieter KranzlmÃŒller, editors, *High Performance Computing and Communications*, volume 4208 of *Lecture Notes in Computer Science*, pages 170–179. Springer Berlin / Heidelberg, 2006.

[53] T.H. Brus, M.C.J.D. van Eekelen, M.O. van Leer, M.J. Plasmeijer, and H.P. Barendregt. CLEAN - A Language for Functional Graph Rewriting. In Kahn, editor, *Proc. of Conference on Functional Programming Languages and Computer Architecture (FPCA '87)*, volume 274 of *Lecture Notes in Computer Science*, pages 364–384, Portland, Oregon, USA, 1987. Springer-Verlag.

[54] G. L. Burn, S. L. Peyton Jones, and J. D. Robson. The spineless G-machine. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 244–258, New York, NY, USA, 1988. ACM.

[55] F. Warren Burton. Annotations to Control Parallelism and Reduction Order in the Distributed Evaluation of Functional Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(2):159–174, 1984.

[56] Russell Butek. Which style of WSDL should I use? *IBM developerWorks*, May 2005. http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/.

[57] Rajkumar Buyya, editor. *High Performance Cluster Computing*, volume 1 and 2. Prentice Hall - PTR, NJ, USA, 1999.

[58] Junwei Cao, Stephen A. Jarvis, Subhash Saini, and Graham R. Nudd. GridFlow: Workflow Management for Grid Computing. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 198, Washington, DC, USA, 2003. IEEE Computer Society.

[59] Luca Cardelli. Basic Polymorphic Typechecking. *Science of Computer Programming*, 8(2), April 1987.

[60] M. Castan, A. Contessa, E. Cousin, C. Coustet, and B. Lecussan. MaRs: a parallel graph reduction multiprocessor. *ACM SIGARCH Computer Architecture News*, 16(3):17–24, 1988.

[61] Girish B. Chafle, Sunil Chandra, Vijay Mann, and Mangala Gowri Nanda. Decentralized orchestration of composite web services. In *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 134–143, New York, NY, USA, 2004. ACM Press.

[62] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data Parallel Haskell: a status report. In *DAMP 2007: Workshop on Declarative Aspects of Multicore Programming*. ACM Press, November 2007.

[63] Mason Chang, Edwin Smith, Rick Reitmaier, Michael Bebenita, Andreas Gal, Christian Wimmer, Brendan Eich, and Michael Franz. Tracing for web 3.0: trace compilation for the next generation web applications. In *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 71–80, New York, NY, USA, 2009. ACM.

[64] Andrew Chien, Brad Calder, Stephen Elbert, and Karan Bhatia. Entropia: Architecture and Performance of an Enterprise Desktop Grid System. *Journal of Parallel and Distributed Computing*, 63(5), May 2003.

[65] Wei-Ngan Chin. Safe fusion of functional expressions. *SIGPLAN Lisp Pointers*, V(1):11–20, 1992.

[66] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. W3C note, World Wide Web Consortium, March 2001.

[67] Alonzo Church. A Set of Postulates for the Foundation of Logic. *The Annals of Mathematics, 2nd Ser.*, 33(2):346–366, April 1932.

[68] Alonzo Church. *The Calculi of Lambda Conversion.* Princeton University Press, Princeton, N.J., 1941.

[69] Alonzo Church and J. Barkley Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, May 1936.

[70] David Churches, Gabor Gombas, Andrew Harrison, Jason Maassen, Craig Robinson, Matthew Shields, Ian Taylor, and Ian Wang. Programming scientific and distributed workflow with Triana services. *Concurrency and Computation: Practice and Experience*, 18(10):1021–1037, August 2006. Special Issue on Grid Workflow.

[71] Chris Clack. Realisations for Non-Strict Languages. In Kevin Hammond and Greg Michelson, editors, *Research Directions in Parallel Functional Programming*, chapter 6, pages 149–187. Springer-Verlag, London, UK, 1999.

[72] Chris Clack and Simon L. Peyton Jones. The four-stroke reduction engine. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 220–232, New York, NY, USA, 1986. ACM Press.

[73] Chris Clack and Simon L. Peyton-Jones. Strictness analysis – a practical approach. In J. P. Jouannaud, editor, *Conference on Functional Programming and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 35–39, Berlin, 1985. Springer-Verlag.

[74] Cluster Resources, Inc. TORQUE Resource Manager. http://www.clusterresources.com/products/torque-resource-manager.php.

[75] Alessandro Contessa. An approach to fault tolerance and error recovery in a parallel graph reduction machine: MaRS - a case study. *SIGARCH Comput. Archit. News*, 16(3):25–32, 1988.

[76] Peter Couvares, Tevfik Kosar, Alain Roy, Jeff Weber, and Kent Wenger. Workflow Management in Condor. In I.J. Taylor, E. Deelman, D. Gannon, and M. Shields, editors, *Workflows for eScience - Scientific Workflows for Grids*, pages 357–375. Springer, December 2006.

[77] M. Crispin. Internet Message Access Protocol - Version 4rev1, March 2003. RFC 3501.

[78] Doug Cutting and Eric Baldeschwieler. Meet Hadoop. In *OSCON*, Portland, OR, USA, July 2007.

[79] John Darlington and Mike Reeve. ALICE: a multi-processor reduction machine for the parallel evaluation of applicative languages. In *FPCA '81: Proceedings of the 1981 conference on Functional programming languages and computer architecture*, pages 65–76, New York, NY, USA, 1981. ACM.

[80] Alan L. Davis and Robert M. Keller. Data Flow Program Graphs. *Computer*, 15(2):26–41, 1982.

[81] Umeshwar Dayal, Meichun Hsu, and Rivka Ladin. Business Process Coordination: State of the Art, Trends, and Open Issues. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 3–13, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[82] Marc de Kruijf and Karthikeyan Sankaralingam. MapReduce for the Cell B.E. Architecture. Technical Report CS-TR-2007-1625, University of Wisconsin Computer Sciences, October 2007.

[83] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, December 2004.

[84] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Sonal Patil, Mei-Hui Su, Karan Vahi, and Miron Livny. Pegasus: Mapping Scientific Workflows onto the Grid. In *Across Grids Conference 2004*, Nicosia, Cyprus, 2004.

[85] Ewa Deelman, Dennis Gannon, Matthew Shields, and Ian Taylor. Workflows and e-Science: An overview of workflow system features and capabilities. *Future Gener. Comput. Syst.*, 25(5):528–540, 2009.

[86] Jack B. Dennis and David P. Misunas. A preliminary architecture for a basic data-flow processor. *SIGARCH Comput. Archit. News*, 3(4):126–132, 1974.

[87] Frank DeRemer and Hans Kron. Programming-in-the large versus programming-in-the-small. In *Proceedings of the international conference on Reliable software*, pages 114–121, New York, NY, USA, 1975. ACM.

[88] Julian Dermoudy. *Effective Run-Time Management of Parallelism in a Functional Programming Context.* PhD thesis, School of Computing, University of Tasmania, March 2002.

[89] Alin Deutsch, Yannis Papakonstantinou, and Yu Xu. The NEXT framework for logical XQuery optimization. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 168–179. VLDB Endowment, 2004.

[90] Jeff Dexter. Combining XQuery and Web Services. *XML Jornal*, January 2005.

[91] Jack Dongarra and Alexey L. Lastovetsky. *High Performance Heterogeneous Computing (Wiley Series on Parallel and Distributed Computing).* Wiley-Interscience, August 2009.

[92] Xing Du and Xiaodong Zhang. Coordinating Parallel Processes on Networks of Workstations. *Journal of Parallel and Distributed Computing*, 46(2):125–135, 1997.

[93] Johann Eder and Walter Liebhart. Workflow Recovery. In *COOPIS '96: Proceedings of the First IFCIS International Conference on Cooperative Information Systems*, page 124, Washington, DC, USA, 1996. IEEE Computer Society.

[94] Wael R. Elwasif, James S. Plank, and Rich Wolski. Data Staging Effects in Wide Area Task Farming Applications. In *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 122, Washington, DC, USA, 2001. IEEE Computer Society.

[95] Robert A. Van Engelen and Kyle A. Gallivan. The gSOAP Toolkit for Web Services and Peer-to-Peer Computing Networks. In *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, page 128, Washington, DC, USA, 2002. IEEE Computer Society.

[96] Theodore Faber, Joe Touch, and Wei Yue. The TIME-WAIT State in TCP and Its Effect on Busy Servers. In *Proceedings of IEEE INFOCOM*, pages 1573–1584, New York, New York, March 1999. IEEE.

[97] Katrina E. Kerry Falkner, Paul D. Coddington, and Michael J. Oudshoorn. Implementing Asynchronous Remote Method Invocation in Java. In *Proc. of Parallel and Real Time Systems (PART'99)*, Melbourne, November 1999.

[98] David C. Fallside and Priscilla Walmsley. XML Schema Part 0: Primer Second Edition. W3C recommendation, World Wide Web Consortium, October 2004.

[99] Leonidas Fegaras, Ranjan Dash, and YingHui Wang. A Fully Pipelined XQuery Processor. In *3rd Int'l Workshop on XQuery Implementation, Experience and Perspectives, Collocated with ACM SIGMOD 2006*, Chicago, USA, June 2006.

[100] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A report on the SISAL language project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, 1990.

[101] Mary Fernandez, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh. XQuery 1.0 and XPath 2.0 Data Model (XDM). W3C recommendation, World Wide Web Consortium, January 2007.

[102] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1, June 1999. RFC 2616.

[103] Roy Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, Irvine, 2000.

[104] Daniela Florescu, Chris Hillery, Donald Kossmann, Paul Lucas, Fabio Riccardi, Till Westmann, J. Carey, and Arvind Sundararajan. The BEA streaming XQuery processor. *The VLDB Journal*, 13(3):294–315, 2004.

[105] I. Foster, H. Kishimoto, A. Savva, D. Berry, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Treadwell, and J. Von Reich. The Open Grid Services Architecture, Version 1.5. GFD 80, Open Grid Forum, July 2006.

[106] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[107] Ian Foster, David R. Kohr, Jr., Rakesh Krishnaiyer, and Alok Choudhary. A library-based approach to task parallelism in a data-parallel language. *J. Parallel Distrib. Comput.*, 45(2):148–158, 1997.

[108] Simon Foster. HAIFA: An XML Based Interoperability Solution for Haskell. In *6th Symposium on Trends in Functional Programming (TFP 2005)*, pages 103–118. Tartu University Press, 2005.

[109] Juliana Freire and Claudio T. Silva. Simplifying the Design of Workflows for Large-Scale Data Exploration and Visualization. In *Microsoft eScience Workshop*, Indianapolis, Indiana, December 2008.

[110] Daniel P. Friedman and David S. Wise. CONS Should Not Evaluate its Arguments. In *Automata, Languages and Programming*, pages 257–284. Edinburgh U. Press, Edinburgh, Scotland, 1976.

[111] Vladimir Gapeyev and Benjamin C. Pierce. Regular Object Types. In *ECOOP 2003 – Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 469–474. Springer Berlin / Heidelberg, July 2003.

[112] Frank Garvan. *The MAPLE Book.* Chapman & Hall, November 2001.

[113] Wolfgang Gentzsch. Sun Grid Engine: Towards Creating a Compute Power Grid. In *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 35, Washington, DC, USA, 2001. IEEE Computer Society.

[114] Yolanda Gil, Paul Groth, Varun Ratnakar, and Christian Fritz. Expressive Reusable Workflow Templates. In *Proceedings of the Fifth IEEE International Conference on e-Science*, Oxford, UK, December 2009.

[115] Michael Gillmann, Ralf Mindermann, and Gerhard Weikum. Benchmarking and Configuration of Workflow Management Systems. In *CooplS '02: Proceedings of the 7th International Conference on Cooperative Information Systems*, pages 186–197, London, UK, 2000. Springer-Verlag.

[116] Stephen Gilmore. Programming in Standard ML '97: A Tutorial Introduction. Technical Report ECS-LFCS-97-364, Laboratory for Foundations of Computer Science, Department of Computer Science, The University of Edinburgh, September 1997.

[117] Hugh Glaser, Chris Hankin, and David Till. *Principles of functional programming.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.

[118] Tristan Glatard and Johan Montagnat. Implementation of Turing machines with the Scufl data-flow language. In *Proceedings of the 3rd International Workshop on Workflow Systems in e-Science (WSES'08)*, Lyon, France, May 2008.

[119] Mudit Goel. Process Networks in Ptolemy II. Technical Memorandum UCB/ERL M98/69, Electronics Research Laboratory, Berkeley, CA 94720, December 1998.

[120] B. Goldberg and P. Hudak. Implementing functional programs on a hypercube multiprocessor. In *Proceedings of the third conference on Hypercube concurrent computers and applications*, pages 489–504, New York, NY, USA, 1988. ACM.

[121] Benjamin Goldberg. Buckwheat: graph reduction on a shared-memory multiprocessor. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 40–51, New York, NY, USA, 1988. ACM Press.

[122] Seth Copen Goldstein, Klaus Erik Schauser, and David E. Culler. Lazy threads: implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, 1996.

[123] Daniel Goodman. Martlet: a scientific work-flow language for abstracted parallisation. In *Proceedings of UK e-science All Hands meeting*, Nottingham, UK, September 2006.

[124] Michael J.C. Gordon. *Programming language theory and its implementation.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1988.

[125] Greg Goth. Critics Say Web Services Need a REST. *IEEE Distributed Systems Online*, 5(12):1, 2004.

[126] Steve Gregory and Martha Paschali. A Prolog-Based Language for Workflow Programming. In *Coordination Models and Languages*, volume 4467 of *Lecture Notes in Computer Science*, pages 56–75. Springer, Berlin / Heidelberg, 2007.

[127] Maxim Grinev and Sergey D. Kuznetsov. Towards an Exhaustive Set of Rewriting Rules for XQuery Optimization: BizQuery Experience. In *ADBIS '02: Proceedings of the 6th East European Conference on Advances in Databases and Information Systems*, pages 340–345, London, UK, 2002. Springer-Verlag.

[128] Maxim Grinev and Peter Pleshachkov. Rewriting-Based Optimization for XQuery Transformational Queries. In *IDEAS '05: Proceedings of the 9th International Database Engineering & Application Symposium*, pages 163–174, Washington, DC, USA, 2005. IEEE Computer Society.

[129] Zhimao Guo, Xiaoling Wang, and Aoying Zhou. WSQuery: XQuery for Web Services Integration. In *Database Systems for Advanced Applications*, volume 3453 of *Lecture Notes in Computer Science*, pages 372–384. Springer Berlin / Heidelberg, 2005.

[130] J. R Gurd, C. C Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Commun. ACM*, 28(1):34–52, 1985.

[131] Claus Hagen and Gustavo Alonso. Exception handling in workflow management systems. *IEEE Transactions on Software Engineering*, 26(10):943–958, October 2000.

[132] K. Hammond, A. Al Zain, G. Cooperman, D. Petcu, and P. Trinder. SymGrid: a Framework for Symbolic Computation on the Grid. In *Proc. EuroPar'07 - European Conf. on Parallel Processing*, volume 4703 of *Lecture Notes in Computer Science*, Rennes, France, August 2007. Spinger-Verlag.

[133] Kevin Hammond. Parallel Functional Programming: An Introduction. In *International Symposium on Parallel Symbolic Computation*, Hagenberg/Linz, Austria, September 1994. World Scientific.

[134] Matthew Harren, Mukund Raghavachari, Oded Shmueli, Michael G. Burke, Rajesh Bordawekar, Igor Pechtchanski, and Vivek Sarkar. XJ: Facilitating XML Processing in Java. In *WWW 2005*, pages 278–287, 2005.

[135] Tim Harris, Simon Marlow, and Simon Peyton Jones. Haskell on a shared-memory multiprocessor. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 49–61, New York, NY, USA, 2005. ACM Press.

[136] P.H. Hartel, H. Glaser, and J. Wild. On the benefits of different analyses in the compilation of a lazy functional language. In *Proceedings of the Workshop on the Parallel Implementation of Functional Languages*, pages 123–145, University of Southampton, 1991.

[137] Thomas Heinis, Cesare Pautasso, and Gustavo Alonso. Design and Evaluation of an Autonomic Workflow Engine. In *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, pages 27–38, Washington, DC, USA, 2005. IEEE Computer Society.

[138] Robert L. Henderson. Job scheduling under the Portable Batch System. In *Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 279–294. Springer Berlin / Heidelberg, 1995.

[139] Tony Hey and Anne Trefethen. e-Science and its implications. *Philos Transact A Math Phys Eng Sci.*, 361:1809–25, August 2003.

[140] Guido Hogen and Rita Loogen. A New Stack Technique for the Management of Runtime Structures in Distributed Implementations. Technical Report 1993-03, Department Of Computer Science, Aachener Informatik-Berichte (AIB), 1993.

[141] David Hollingsworth. The Workflow Reference Model. Specification TC00-1003, Workflow Management Coalition, January 1995. Issue 1.1.

[142] Arnaud Le Hors, Philippe Le Hegaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne. Document Object Model (DOM) Level 2 Core Specification, version 1.0. W3C recommendation, World Wide Web Consortium, November 2000.

[143] Qifeng Huang and Yan Huang. Workflow Engine with Multi-level Parallelism Supports. In *UK e-Science All Hands Meeting*, Nottingham, September 2005.

[144] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys (CSUR)*, 21(3):359–411, 1989.

[145] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A History of Haskell: being lazy with class. In *The Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*, San Diego, California, June 2007.

[146] Paul Hudak and Lauren Smith. Para-functional programming: a paradigm for programming multiprocessor systems. In *POPL '86: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 243–254, New York, NY, USA, 1986. ACM.

[147] Paul Hudak and Raman S. Sundaresh. On the expressiveness of purely-functional I/O systems. Technical Report YALEU/DCS/RR-665, Department of Computer Science, Yale University, March 1989.

[148] Paul Hudak and Jonathan Young. Higher-order strictness analysis in untyped lambda calculus. In *POPL '86: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 97–109, New York, NY, USA, 1986. ACM Press.

[149] R. J. M. Hughes. Super-combinators: A new implementation method for applicative languages. In *LFP '82: Proceedings of the 1982 ACM symposium on LISP and functional programming*, pages 1–10, New York, NY, USA, 1982. ACM.

[150] D. Hull, R. Stevens, and P. Lord. Describing web services for user-oriented retrieval. In *W3C Workshop on Frameworks for Semantics in Web Services*, Innsbruck, Austria, June 2005. Digital Enterprise Research Institute (DERI).

[151] Duncan Hull, Robert Stevens, Phillip Lord, and Carole Goble. Integrating bioinformatics resources using shims. In *Twelfth International conference on Intelligent Systems for Molecular Biology (ISMB2004)*, Glasgow, UK, 2004.

[152] Duncan Hull, Robert Stevens, Phillip Lord, Chris Wroe, and Carole Goble. Treating shimantic web syndrome with ontologies. In *First Advanced Knowledge Technologies workshop on Semantic Web Services (AKT-SWS04)*. KMi, The Open University, Milton Keynes, UK, 2004. (See Workshop proceedings CEUR-WS.org (ISSN:16130073) Volume 122 - AKT-SWS04).

[153] Christopher Hylands, Edward Lee, Jie Liu, Xiaojun Liu, Stephen Neuendorffer, Yuhong Xiong, Yang Zhao, and Haiyang Zheng. Overview of the Ptolemy Project. Technical Memorandum UCB/ERL M03/25, University of California, Berkeley, CA, 94720, USA, July 2003.

[154] David Jackson, Quinn Snell, and Mark Clement. Core Algorithms of the Maui Scheduler. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 87–102. Springer Verlag, 2001. Lect. Notes Comput. Sci. vol. 2221.

[155] Jr. James S. Mattson and William G. Griswold. Speculative Evaluation for Parallel Graph Reduction. In *PACT '94: Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques*, pages 331–334, Amsterdam, The Netherlands, The Netherlands, 1994. North-Holland Publishing Co.

[156] Philipp K. Janert. *Gnuplot in Action: Understanding Data with Graphs.* Manning Publications Co., August 2009.

[157] Thomas Johnsson. Efficient compilation of lazy evaluation. In *SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, pages 58–69, New York, NY, USA, 1984. ACM Press.

[158] Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 190–203, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[159] Thomas Johnsson. Target code generation from G-machine code. In *Proc. of a workshop on Graph reduction*, pages 119–159, London, UK, 1987. Springer-Verlag.

[160] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances In Dataflow Programming Languages. *ACM Computing Surveys (CSUR)*, 36(1):1–34, 2004.

[161] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[162] Simon L. Peyton Jones, Chris D. Clack, Jon Salkild, and Mark Hardie. GRIP - A high-performance architecture for parallel graph reduction. In *Proceedings of the Functional Programming Languages and Computer Architecture*, pages 98–112, London, UK, 1987. Springer-Verlag.

[163] Simon L Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Phil Wadler. The Glasgow Haskell compiler: a technical overview. In *Proceedings of Joint Framework for Information Technology Technical Conference*, pages 249–257, Keele, March 1993.

[164] Simon L. Peyton Jones and Jon Salkild. The spineless tagless G-machine. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 184–201, New York, NY, USA, 1989. ACM Press.

[165] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 71–84, New York, NY, USA, 1993. ACM Press.

[166] Simon Peyton Jones. Foreword. In Kevin Hammond and Greg Michelson, editors, *Research Directions in Parallel Functional Programming*, pages xiii–xv. Springer-Verlag, London, UK, 1999.

[167] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In Tony Hoare, Manfred Broy, and Ralf Steinbruggen, editors, *Engineering theories of software construction*, pages 47–96. IOS Press, 2001. Presented at the 2000 Marktoberdorf Summer School.

[168] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, January 2003.

[169] Simon Peyton Jones and David Lester. *Implementing functional languages: a tutorial.* Prentice Hall, 1992.

[170] Guy Lewis Steele Jr. and Gerald Jay Sussman. Lambda: The Ultimate Imperative. AI Lab Memo AIM-353, MIT AI Lab, March 1976.

[171] Sahalu Junaidu, Antony J. T. Davie, and Kevin Hammond. Naira: A Parallel Haskell Compiler. In Chris Clack, Kevin Hammond, and Antony J. T. Davie, editors, *Implementation of Functional Languages, 9th International Workshop, IFL 1997, St. Andrews, Scotland, UK, September 10-12, 1997, Selected Papers*, volume 1467 of *Lecture Notes in Computer Science*, pages 214–230. Springer, 1998.

[172] Sahalu B. Junaidu. *A Parallel Functional Language Compiler for Message-Passing Multicomputers.* PhD thesis, Division of Computing Science, University of St Andrews, March 1998.

[173] Pekka Kanerva. State of the art of SOAP libraries in Python and Ruby. Technical Report 2007-02, Helsinki Institute for Information Technology, August 2007.

[174] O. Kaser, S. Pawagi, C. R. Ramakrishnan, I. V. Ramakrishnan, and R. C. Sekar. Fast parallel implementation of lazy languages – the EQUALS experience. In *LFP '92: Proceedings of the 1992 ACM conference on LISP and functional programming*, pages 335–344, New York, NY, USA, 1992. ACM Press.

[175] O. Kaser, C.R. Ramakrishnan, and R. Sekar. A High Performance Runtime System for Parallel Evaluation of Lazy languages. In *Int. Symp. on Parallel Symbolic Computation (PASCO)*, September 1994.

[176] James H. Kaufman, Tobin J. Lehman, Glenn Deen, and John Thomas. OptimalGrid – autonomic computing on the Grid. *IBM developerWorks*, June 2003.

[177] Michael Kay. Saxon: Anatomy of an XSLT processor. *IBM developerWorks*, February 2001.

[178] Michael Kay. XSLT and XPath Optimization. In *XML Europe 2004*, Amsterdam, 2004.

[179] Michael Kay. XSL Transformations (XSLT) Version 2.0. W3C recommendation, World Wide Web Consortium, January 2007.

[180] Michael Kay. Ten Reasons Why Saxon XQuery is Fast. *IEEE Data Engineering Bulletin*, 31(4):65–74, 2008.

[181] Peter M. Kelly, Paul D. Coddington, and Andrew L. Wendelborn. Distributed, parallel web service orchestration using XSLT. In *1st IEEE International Conference on e-Science and Grid Computing*, Melbourne, Australia, December 2005.

[182] Peter M. Kelly, Paul D. Coddington, and Andrew L. Wendelborn. Compilation of XSLT into Dataflow Graphs for Web Service Composition. In *6th IEEE International Symposium on Cluster Computing and the Grid (CCGrid06)*, Singapore, May 2006.

[183] Peter M. Kelly, Paul D. Coddington, and Andrew L. Wendelborn. A Simplified Approach to Web Service Development. In *4th Australasian Symposium on Grid Computing and e-Research (AusGrid 2006)*, Hobart, Australia, January 2006.

[184] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, 11(1), August 1998.

[185] G Kiczales. Towards a New Model of Abstraction in the Engineering of Software. In *Proc. of Intl. Workshop on New Models for Software Architecture (IMSA): Reflection and Meta-Level Architecture*, 1992.

[186] Hugh Kingdon, David R. Lester, and Geoffrey L. Burn. The HDG-machine: a highly distributed graph-reducer for a transputer network. *Comput. J.*, 34(4):290–301, 1991.

[187] Werner Kluge. Realisations for Strict Languages. In Kevin Hammond and Greg Michelson, editors, *Research Directions in Parallel Functional Programming*, chapter 5, pages 121–148. Springer-Verlag, London, UK, 1999.

[188] Ulrike Klusik, Rita Loogen, and Steffen Priebe. Controlling parallelism and data distribution in Eden. In *Selected papers from the 2nd Scottish Functional Programming Workshop (SFP00)*, pages 53–64, Exeter, UK, UK, 2000. Intellect Books.

[189] Doug Kohlert and Arun Gupta. The Java API for XML-Based Web Services (JAX-WS) 2.1, May 2007. Maintenance Release.

[190] Tevfik Kosar and Miron Livny. Stork: Making Data Placement a First Class Citizen in the Grid. In *Proceedings of 24th IEEE Int. Conference on Distributed Computing Systems (ICDCS2004)*, Tokyo, Japan, 2004.

[191] Pavel Kulchenko. The Evolution of SOAP::LITE. *ONLamp.com*, June 2001.

[192] Ralf Lämmel and Erik Meijer. Revealing the X/O Impedance Mismatch (Changing Lead into Gold). In *Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, pages 285–367. Springer Berlin / Heidelberg, June 2007.

[193] K.G. Langendoen. *Graph reduction on shared-memory multiprocessors*. PhD thesis, Department of Computer Systems, University of Amsterdam, April 1993.

[194] Mario Lauria and Andrew Chien. MPI-FM: high performance MPI on workstation clusters. *J. Parallel Distrib. Comput.*, 40(1):4–18, 1997.

[195] Daniel E. Lenoski and Wolf-Dietrich Weber. *Scalable Shared-Memory Multiprocessing*. Morgan Kaufmann, June 1995.

[196] Ehud Shapiro Leon Sterling. *The Art of Prolog, Second Edition: Advanced Programming Techniques (Logic Programming)*. The MIT Press, March 1994.

[197] Claudia Leopold. *Parallel and Distributed Computing: A Survey of Models, Paradigms and Approaches*. Wiley-Interscience, November 2000.

[198] Frank Leymann. Web Services Flow Language (WSFL 1.0). Technical report, IBM, May 2001.

[199] Jing Li, Huibiao Zhu, and Geguang Pu. Conformance Validation between Choreography and Orchestration. In *TASE '07: Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*, pages 473–482, Washington, DC, USA, 2007. IEEE Computer Society.

[200] Peter Li, Juan Castrillo, Giles Velarde, Ingo Wassink, Stian Soiland-Reyes, Stuart Owen, David Withers, Tom Oinn, Matthew Pocock, Carole Goble, Stephen Oliver, and Douglas Kell. Performing statistical analyses on quantitative data in Taverna workflows: An example using R and maxdBrowse to identify differentially-expressed genes from microarray data. *BMC Bioinformatics*, 9(1):334, 2008.

[201] Chua Ching Lian, Francis Tang, Praveen Issac, and Arun Krishnan. GEL: Grid Execution Language. *J. Parallel Distrib. Comput.*, 65(7):857–869, 2005.

[202] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification, Second Edition*. Prentice Hall, April 1999.

[203] David T. Liu and Michael J. Franklin. GridDB: a data-centric overlay for scientific grids. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 600–611. VLDB Endowment, 2004.

[204] Jiuxing Liu, Balasubramanian Chandrasekaran, Jiesheng Wu, Weihang Jiang, Sushmitha Kini, Weikuan Yu, Darius Buntinas, Peter Wyckoff, and D K. Panda. Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 58, Washington, DC, USA, 2003. IEEE Computer Society.

[205] Zhen Hua Liu, Muralidhar Krishnaprasad, and Vikas Arora. Native XQuery processing in Oracle XMLDB. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 828–833, New York, NY, USA, 2005. ACM.

[206] H.-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Peña, S. Priebe, Á J. Rebón, and P. W. Trinder. Comparing Parallel Functional Languages: Programming and Performance. *Higher-Order and Symbolic Computation*, 16(3):203–251, 2003.

[207] Hans Wolfgang Loidl. *Granularity in Large-Scale Parallel Functional Programming.* PhD thesis, Department of Computing Science, University of Glasgow, March 1998.

[208] Rita Loogen, Herbert Kuchen, Klaus Indermark, and Werner Damm. Distributed Implementation of Programmed Graph Reduction. In *Conference on Parallel Architectures and Languages Europe (PARLE) '89*, volume 365 of *Lecture Notes in Computer Science*, 1989.

[209] B. Ludascher and I. Altintas. On Providing Declarative Design and Programming Constructs for Scientific Workflows based on Process Networks. Technical Note SciDAC-SPA-TN-2003-01, San Diego Supercomputer Center, UC San Diego, 2003.

[210] Bertram Ludascher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice & Experience*, 18(10):1039–1065, 2006.

[211] Bertram Ludascher, Mathias Weske, Timothy McPhillips, and Shawn Bowers. Scientific Workflows: Business as Usual? *Business Process Management*, 5701, 2009.

[212] Carol M. Lushbough, Michael K. Bergman, Carolyn J. Lawrence, Doug Jennewein, and Volker Brendel. Implementing bioinformatic workflows within the BioExtract Server. *International Journal of Computational Biology and Drug Design*, 1(3):302–312, 2008.

[213] Akshay Luther, Rajkumar Buyya, Rajiv Ranjan, and Srikumar Venugopal. Peer-to-Peer Grid Computing and a .NET-based Alchemi Framework. In Laurence Yang and Minyi Guo, editors, *High Performance Computing: Paradigm and Infrastructure.* Wiley Press, Fall 2004.

[214] Ashok Malhotra, Jim Melton, and Norman Walsh. XQuery 1.0 and XPath 2.0 Functions and Operators. W3C recommendation, World Wide Web Consortium, January 2007.

[215] Benoit B. Mandelbrot. *Fractals and Chaos: The Mandelbrot Set and Beyond.* Springer, January 2004.

[216] Stefan Manegold. An empirical evaluation of XQuery processors. *Information Systems*, 33(2):203–220, 2008.

[217] Luc Maranget. GAML: a parallel implementation of Lazy ML. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 102–123, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[218] Paul N. Martinaitis, Craig J. Patten, and Andrew L. Wendelborn. Component-based stream processing "in the cloud". In *CBHPC '09: Proceedings of the 2009 Workshop on Component-Based High Performance Computing*, pages 1–12, New York, NY, USA, 2009. ACM.

[219] Paul N. Martinaitis and Andrew L. Wendelborn. Representing eager evaluation in a demand driven model of streams on cloud infrastructure. In *CCGrid 2010: The 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2010.

[220] Karen M. McCann, Maurice Yarrow, Adrian DeVivo, and Piyush Mehrotra. ScyFlow: an environment for the visual specification and execution of scientific workflows. *Concurrency and Computation: Practice and Experience*, 18(10):1155–1167, August 2006. Special Issue on Grid Workflow.

[221] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3(4):184–195, 1960.

[222] Stephen McGough, Laurie Young, Ali Afzal, Steven Newhouse, and John Darlington. Workflow Enactment in ICENI. In *UK e-Science All Hands Meeting (AHM 2004)*, Nottingham, UK, August 2004.

[223] James McGovern, Sameer Tyagi, Michael Stevens, and Sunil Mathew. Service Oriented Architecture. In *Java Web Services Architecture*, chapter 2. Elsevier Science, July 2003.

[224] Craig McMurtry, Marc Mercuri, Nigel Watling, and Matt Winkler. *Windows Communication Foundation Unleashed (WCF) (Unleashed).* Sams, Indianapolis, IN, USA, 2007.

[225] T. McPhillips, S. Bowers, and B. Ludascher. Collection-Oriented Scientific Workflows for Integrating and Analyzing Biological Data. In *3rd International Conference on Data Integration for the Life Sciences (DILS)*, LNCS/LNBI, 2006.

[226] Erik Meijer and Brian Beckman. XLinq: XML Programming Refactored (The Return Of The Monoids). In *Proc. XML 2005*, Atlanta, Georgia, USA, November 2005.

[227] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706, New York, NY, USA, 2006. ACM.

[228] Erik Meijer, Wolfram Schulte, and Gavin Bierman. Programming with circles, triangles and rectangles. In *Proceedings of XML 2003*, 2003.

[229] Message Passing Interface Forum. The Message Passing Interface. *International Journal of Supercomputing Applications*, 8(3/4), Fall/Winter 1994.

[230] Luiz A.V.C. Meyer, Shaila C. Rossle, Paulo M. Bisch, and Marta Mattoso. Parallelism in Bioinformatics Workflows. In *High Performance Computing for Computational Science - VECPAR 2004*, volume 3402 of *Lecture Notes in Computer Science*, pages 583–597. Springer Berlin / Heidelberg, 2005.

[231] Greg Michaelson. *An Introduction to Functional Programming through Lambda Calculus.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[232] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[233] M. Mosconi and M Porta. Iteration constructs in data-flow visual programming languages. *Computer Languages*, 26(2-4):67–104, 2000.

[234] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[235] Alan Mycroft. Polymorphic type schemes and recursive definitions. In G. Goos and J. Hartmanis, editors, *International Symposium on Programming - 6th Colloquium*, volume 167 of *Lecture Notes in Computer Science*, Toulouse, France, April 1984. Springer-Verlag.

[236] J. Myers and M. Rose. Post Office Protocol - Version 3, May 1996. RFC 1939.

[237] Mangala Gowri Nanda, Satish Chandra, and Vivek Sarkar. Decentralizing execution of composite web services. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 170–187, New York, NY, USA, 2004. ACM Press.

[238] Pat Niemeyer. The BeanShell User Manual - version 1.3. http://www.beanshell.org, 2002.

[239] OASIS. Introduction to UDDI: Important Features and Concepts, October 2004. http://uddi.org/pubs/uddi-tech-wp.pdf.

[240] Object Management Group. The Common Object Request Broker: Architecture and Specification (CORBA), revision 2.0, 1995.

[241] John T. O'Donnell. Dialogues: A basis for constructing programming environments. In *Proceedings of the ACM SIGPLAN 85 symposium on Language issues in programming environments*, pages 19–27, New York, NY, USA, 1985. ACM.

[242] Tom Oinn, Mark Greenwood, Matthew Addis, M. Nedim Alpdemir, Justin Ferris, Kevin Glover, Carole Goble, Antoon Goderis, Duncan Hull, Darren Marvin, Peter Li, Phillip Lord, Matthew R. Pocock, Martin Senger, Robert Stevens, Anil Wipat, and Chris Wroe. Taverna: Lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, August 2006. Special Issue on Grid Workflow.

[243] Nicola Onose and Jerome Simeon. XQuery at your web service. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 603–611, New York, NY, USA, 2004. ACM Press.

[244] Shankar Pal, Istvan Cseri, Oliver Seeliger, Michael Rys, Gideon Schaller, Wei Yu, Dragan Tomic, Adrian Baras, Brandon Berg, Denis Churin, and Eugene Kogan. XQuery implementation in a relational database system. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 1175–1186. VLDB Endowment, 2005.

[245] Steven Gregory Parker. *The SCIRun problem solving environment and computational steering software system.* PhD thesis, Department of Computer Science, The University of Utah, 1999. Adviser-Johnson, Christopher R.

[246] Linda Dailey Paulson. Building Rich Web Applications with Ajax. *Computer*, 38(10):14–17, 2005.

[247] Cesare Pautasso and Gustavo Alonso. The JOpera Visual Composition Language. *Journal of Visual Languages and Computing (JVLC)*, 16(1-2):119–152, 2005.

[248] Cesare Pautasso and Gustavo Alonso. Parallel Computing Patterns for Grid Workflows. In *Proceedings of the HPDC2006 workshop on Workflows in support for large-scale Science (WORKS06)*, Paris, France, 2006.

[249] Cesare Pautasso, Thomas Heinis, and Gustavo Alonso. JOpera: Autonomic Service Orchestration. *IEEE Data Engineering Bulletin*, 29, September 2006.

[250] Chris Peltz. Web Services Orchestration and Choreography. *Computer*, 36(10):46–52, October 2003.

[251] Srinath Perera, Chathura Herath, Jaliya Ekanayake, Eran Chinthaka, Ajith Ranabahu, Deepal Jayasinghe, Sanjiva Weerawarana, and Glen Daniels. Axis2, Middleware for Next Generation Web Services. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services*, pages 833–840, Washington, DC, USA, 2006. IEEE Computer Society.

[252] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168.

[253] Zongyan Qiu, Xiangpeng Zhao, Chao Cai, and Hongli Yang. Towards the theoretical foundation of choreography. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 973–982, New York, NY, USA, 2007. ACM.

[254] U Radetzki, U Leser, SC Schulze-Rauschenbach, J Zimmermann, J LÃŒssem, T Bode, and AB. Cremers. Adapters, shims, and glue–service interoperability for in silico experiments. *Bioinformatics*, 22(9):1137–43, May 2006.

[255] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[256] Randy J. Ray and Pavel Kulchenko. REST: Representational State Transfer. In *Programming Web Services with Perl*, chapter 11. O'Reilly Media, Inc, December 2002.

[257] Christopher Re, Jérôme Siméon, and Mary F. Fernández. A Complete and Efficient Algebraic Compiler for XQuery. In *ICDE*, page 14, 2006.

[258] Leonard Richardson and Sam Ruby. *RESTful Web Services.* O'Reilly Media, May 2007.

[259] Paul Roe. *Parallel Programming using Functional Languages.* PhD thesis, Department of Computing Science, University of Glasgow, 1991.

[260] Steve Ross-Talbot. Web Services Choreography: Building Peer-to-Peer Relationships. In *Workflows management: new abilities for the biological information overflow*, Naples, Italy, October 2005.

[261] Patrick M. Sansom and Simon L. Peyton Jones. Generational garbage collection for Haskell. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 106–116, New York, NY, USA, 1993. ACM.

[262] Luis F. G. Sarmenta. *Volunteer Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 2001.

[263] Mark Scheevel. NORMA: a graph reduction processor. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 212–219, New York, NY, USA, 1986. ACM Press.

[264] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: a benchmark for XML data management. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 974–985. VLDB Endowment, 2002.

[265] David A. Schmidt. *Denotational semantics: a methodology for language development.* William C. Brown Publishers, Dubuque, IA, USA, 1986.

[266] Maik Schmidt. Ruby as enterprise glue. *Linux Journal*, 2006(147):4, 2006.

[267] R. Sekar, I. V. Ramakrishnan, and P. Mishra. On the power and limitations of strictness analysis. *J. ACM*, 44(3):505–525, 1997.

[268] R. C. Sekar, Shaunak Pawagi, and I. V. Ramarkrishnan. Small domains spell fast strictness analysis. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 169–183, New York, NY, USA, 1990. ACM Press.

[269] Junya Seo, Yoshiyuki Kido, Shigeto Seno, Yoichi Takenaka, and Hideo Matsuda. A Method for Efficient Execution of Bioinformatics Workflows. In *The 20th International Conference on Genome Informatics*, Japan, December 2009.

[270] Roger Sessions. *COM and DCOM: Microsoft's vision for distributed objects.* John Wiley & Sons, Inc., 1998.

[271] Srinath Shankar, Ameet Kini, David J. DeWitt, and Jeffrey Naughton. Integrating databases and workflow systems. *ACM SIGMOD Record*, 34(3):5–11, 2005.

[272] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. A survey of data provenance in e-science. *SIGMOD Rec.*, 34:31–36, September 2005.

[273] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. A Framework for Collecting Provenance in Data-Centric Scientific Workflows. *Web Services, IEEE International Conference on*, 0:427–436, 2006.

[274] Sjaak Smetsers, Erik Barendsen, Marko C. J. D. van Eekelen, and Marinus J. Plasmeijer. Guaranteeing Safe Destructive Updates Through a Type System with Uniqueness Information for Graphs. In *Proceedings of the International Workshop on Graph Transformations in Computer Science*, pages 358–379, London, UK, 1994. Springer-Verlag.

[275] Arnold W. M. Smeulders, Marcel Worring, Simone Santini, Amarnath Gupta, and Ramesh Jain. Content-Based Image Retrieval at the End of the Early Years. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(12):1349–1380, 2000.

[276] Ian M. Smith and D. V. Griffiths. *Programming the Finite Element Method.* Wiley, November 2004. 4th edition.

[277] James Snell, Doug Tidwell, and Pavel Kulchenko. *Programming Web Services with SOAP*. O'Reilly Media, December 2001.

[278] Patricia Gomes Soares. On remote procedure call. In *CASCON '92: Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research*, pages 215–267. IBM Press, 1992.

[279] Jacek Sroka, Grzegorz Kaczor, Jerzy Tyszkiewicz, and Andrzej M. Kierzek. XQTav: an XQuery processor for Taverna environment. *Bioinformatics*, 22(10):1280–1281, 2006.

[280] Guy Steele. *Common LISP. The Language. Second Edition.* Digital Press, June 1990.

[281] W. Richard Stevens. TCP Bulk Data Flow. In *TCP/IP Illustrated, Volume 1: The Protocols*, chapter 20, pages 275–296. Addison-Wesley, 1994.

[282] W. Richard Stevens. TCP Connection Establishment and Termination. In *TCP/IP Illustrated, Volume 1: The Protocols*, chapter 18, pages 229–262. Addison-Wesley, 1994.

[283] W. Richard Stevens. *UNIX Network Programming: Networking APIs: Sockets and XTI; Volume 1.* Prentice Hall PTR; 2nd edition, January 1998.

[284] Sun Microsystems. RPC: Remote Procedure Call Specification, 1988. RFC 1050.

[285] Sun Microsystems. Java remote method invocation specification, October 1998.

[286] V. Sunderam, J. Dongarra, A. Geist, , and R Manchek. The PVM Concurrent Computing System: Evolution, Experiences, and Trends. *Parallel Computing*, 20(4):531–547, April 1994.

[287] Gerald Jay Sussman and Guy Lewis Steele Jr. Scheme: An Interpreter for Extended Lambda Calculus. AI Lab Memo AIM-349, MIT AI Lab, December 1975.

[288] M. Szomszor, T. R. Payne, and L. Moreau. Dynamic Discovery of Composable Type Adapters for Practical Web Services Workflow. In *UK e-Science All Hands Meeting 2006*, Nottingham, September 2006.

[289] Wei Tan, Paolo Missier, Ravi Madduri, and Ian Foster. Building Scientific Workflow with Taverna and BPEL: A Comparative Study in caGrid. In *Service-Oriented Computing — ICSOC 2008 Workshops: ICSOC 2008 International Workshops, Sydney, Australia, December 1st, 2008, Revised Selected Papers*, pages 118–129, Berlin, Heidelberg, 2009. Springer-Verlag.

[290] Andrew S. Tanenbaum and Robbert van Renesse. A critique of the remote procedure call paradigm. In R. Speth, editor, *Proceedings of the EUTECO 88 Conference*, pages 775–783, Vienna, Austria, April 1988. Elsevier Science Publishers B. V. (North-Holland).

[291] Francis Tang, Ching Lian Chua, Liang-Yoong Ho, Yun Ping Lim, Praveen Issac, and Arun Krishnan. Wildfire: distributed, Grid-enabled workflow construction and execution. *BMC Bioinformatics*, 6(69), March 2005.

[292] B. H. Tay and A. L. Ananda. A survey of remote procedure calls. *SIGOPS Oper. Syst. Rev.*, 24(3):68–79, 1990.

[293] Frank Taylor. *Parallel Functional Programming by Partitioning.* PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, University of London, January 1997.

[294] Ian Taylor, Matthew Shields, Ian Wang, and Andrew Harrison. The Triana Workflow Environment: Architecture and Applications. In I.J. Taylor, E. Deelman, D. Gannon, and M. Shields, editors, *Workflows for eScience - Scientific Workflows for Grids*, pages 320–339. Springer, December 2006.

[295] I.J. Taylor, E. Deelman, D. Gannon, and M. Shields, editors. *Workflows for eScience - Scientific Workflows for Grids.* Springer, December 2006.

[296] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed Computing in Practice: The Condor Experience. *Concurrency and Computation: Practice and Experience*, 2004.

[297] S. Thatte. XLANG Web Services for Business Process Design, 2002.

[298] Rafael Tolosana-Calasanz, José A. Bañares, Omer F. Rana, Pedro Álvarez, Joaquín Ezpeleta, and Andreas Hoheisel. Adaptive exception handling for scientific workflows. *Concurrency and Computation: Practice and Experience*, 22:617–642, April 2010.

[299] G. Tremblay and G.R. Gao. The impact of laziness on parallelism and the limits of strictness analysis. In A.P. Wim Bohm and John T. Feo, editors, *Proceedings High Performance Functional Computing*, pages 119–133, Lawrence Livermore National Laboratory CONF-9504126, April 1995.

[300] P. W. Trinder, K. Hammond, Jr. J. S. Mattson, A. S. Partridge, and S. L. Peyton Jones. GUM: a portable parallel implementation of Haskell. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 79–88, New York, NY, USA, 1996. ACM Press.

[301] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + strategy = parallelism. *J. Funct. Program.*, 8(1):23–60, 1998.

[302] Philip W. Trinder, Ed. Barry Jr., M. Kei Davis, Kevin Hammond, Sahalu B. Junaidu, Ulrike Klusik, Hans-Wolfgang Loidl, , and Simon L. Peyton Jones. GpH: An Architecture-Independent Functional Language. Submitted to IEEE Transactions on Software Engineering, special issue on Architecture-Independent Languages and Software Tools for Parallel Processing, July 1998.

[303] P.W. Trinder, H-W. Loidl, and R.F. Pointon. Parallel and Distributed Haskells. *Journal of Functional Programming*, 12(4&5):469–510, 2002.

[304] Hong-Linh Truong, Peter Brunner, Thomas Fahringer, Francesco Nerieri, Robert Samborski, Bartosz Balis, Marian Bubak, and Kuba Rozkwitalski. K-WfGrid Distributed Monitoring and Performance Analysis Services for Workflows in the Grid. In *Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, E-SCIENCE '06, pages 15–, Washington, DC, USA, 2006. IEEE Computer Society.

[305] Aphrodite Tsalgatidou and Thomi Pilioura. An Overview of Standards and Related Technology in Web Services. *Distrib. Parallel Databases*, 12(2-3):135–162, 2002.

[306] Daniele Turi, Paolo Missier, Carole Goble, David De Roure, and Tom Oinn. Taverna Workflows: Syntax and Semantics. In *Proceedings of eScience 2007*, Bangalore, India, December 2007.

[307] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[308] David Turner. A New Implementation Technique for Applicative Languages. *Software, practice & experience (0038-0644)*, 9(1), 1979.

[309] David Turner. An overview of Miranda. *ACM SIGPLAN Notices*, 21(12):158–166, 1986.

[310] David Turner. Church's Thesis and Functional Programming. In A. Olszewski, editor, *Church's Thesis after 70 years*, pages 518–544. Ontos Verlag, Berlin, 2006.

[311] W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.

[312] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, July 2003.

[313] Daniel Veillard. The XML C parser and toolkit of Gnome (libxml). http://www.xmlsoft.org/.

[314] Eelco Visser. Stratego: A Language for Program Transformation Based on Rewriting Strategies. In Aart Middeldorp, editor, *Rewriting Techniques and Applications, 12th International Conference, RTA 2001, Utrecht, The Netherlands, May 22-24, 2001, Proceedings*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–362. Springer, 2001.

[315] Markus Voelter, Michael Kircher, Uwe Zdun, and Michael Englbrecht. Patterns for Asynchronous Invocations in Distributed Object Frameworks. In *EuroPLoP 2003 conference*, Kloster Irsee, Germany, June 2003.

[316] Gregor von Laszewski, Kaizar Amin, Mihael Hategan, Nestor J. Zaluzec, Shawn Hampton, and Al Rossi. GridAnt: A Client-Controllable Grid Workflow System. In *37th Hawai'i International Conference on System Science*, Island of Hawaii, Big Island, January 2004.

[317] Gregor von Laszewski, Mihael Hategan, and Deepti Kodeboyina. Java CoG Kit Workflow. In I.J. Taylor, E. Deelman, D. Gannon, and M. Shields, editors, *Workflows for eScience - Scientific Workflows for Grids*, pages 340–356. Springer, December 2006.

[318] Gregor von Laszewski, Mihael Hategan, and Deepti Kodeboyina. Work Coordination for Grid Computing. In M.P. Bekakos and G.A. Gravvanis, editors, *Grid Technologies: Emerging from Distributed Architectures to Virtual Organizations*, volume 5 of *Advances in Management Information*, pages 309–329. WIT Press, 2006.

[319] Philip Wadler. Deforestation: transforming programs to eliminate trees. In *Proceedings of the Second European Symposium on Programming*, pages 231–248, Amsterdam, The Netherlands, The Netherlands, 1988. North-Holland Publishing Co.

[320] Ueli Wahli, Owen Burroughs, Owen Cline, Alec Go, and Larry Tung. *Web Services Handbook for WebSphere Application Server 6.1.* IBM Redbooks, October 2006.

[321] I. Wassink, P.E. van der Vet, K. Wolstencroft, P.B.T. Neerincx, M. Roos, H. Rauwerda, and T.M. Breit. Analysing scientific workflows: why workflows not only connect web services. In LJ. Zhang, editor, *IEEE Congress on Services 2009*, pages 314–321, Los Alamitos, July 2009. IEEE Computer Society Press.

[322] Darren Webb, Andrew Wendelborn, and Julien Vayssiere. A study of computational reconfiguration in a process network. In *Proceedings of IDEA '00*, February 2000.

[323] Andrew L. Wendelborn and Darren Webb. The PAGIS Grid Application Environment. In P. Sloot et al., editor, *International Conference on Computational Science (ICCS 2003)*, volume 2659 of *Lecture Notes in Computer Science*, pages 1113–1122, Melbourne, June 2003. Springer.

[324] Stephen Wolfram. *The Mathematica Book, Fourth Edition.* Cambridge University Press, March 1999.

[325] Chris Wroe, Carole Goble, Mark Greenwood, Phillip Lord, Simon Miles, Juri Papay, Terry Payne, and Luc Moreau. Automating Experiments Using Semantic Data on a Bioinformatics Grid. *IEEE Intelligent Systems*, 19(1):48–55, 2004.

[326] Hongli Yang, Xiangpeng Zhao, Zongyan Qiu, Geguang Pu, and Shuling Wang. A Formal Model forWeb Service Choreography Description Language (WS-CDL). In *ICWS '06: Proceedings of the IEEE International Conference on Web Services*, pages 893–894, Washington, DC, USA, 2006. IEEE Computer Society.

[327] Ustun Yildiz, Adnene Guabtni, and Anne H. H. Ngu. Towards scientific workflow patterns. In *WORKS '09: Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, pages 1–10, New York, NY, USA, 2009. ACM.

[328] Jia Yu and Rajkumar Buyya. A Novel Architecture for Realizing Grid Workflow using Tuple Spaces. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID 2004)*, Pittsburgh, USA, November 2004. IEEE Computer Society Press, Los Alamitos, CA, USA.

[329] Jia Yu and Rajkumar Buyya. A Taxonomy of Workflow Management Systems for Grid Computing. *Journal of Grid Computing*, 3(3–4):171–200, September 2005.

[330] A. Al Zain, J. Berthold, K. Hammond, and P. Trinder. Orchestrating Production Computer Algebra Components into Portable Parallel Programs. In *Open Source Grid and Cluster Conference*, May 2008.

[331] A. Al Zain, P.W. Trinder, H.W. Loidl, and G.J. Michaelson. Supporting High-Level Grid Parallel Programming: the Design and Implementation of Grid-GUM2. In *UK e-Science Programme All Hands Meeting (AHM)*, September 2007.

[332] A. Al Zain, P.W. Trinder, G.J. Michaelson, and H-W. Loidl. Evaluating a High-Level Parallel Language (GpH) for Computational GRIDs. *IEEE Transactions on Parallel and Distributed Systems*, 19(2):219–233, February 2008.

[333] A. D. Al Zain, P. W. Trinder, G.J.Michaelson, and H-W. Loidl. Managing Heterogeneity in a Grid Parallel Haskell. *Scalable Computing: Practice and Experience*, 7(3), September 2006.

[334] A. D. Al Zain, P. W. Trinder, K. Hammond, A. Konovalov, S. Linton, and J. Berthold. Parallelism without Pain: Orchestrating Computational Algebra Components into a High-Performance Parallel System. In *ISPA '08: Proceedings of the 2008 IEEE International Symposium on Parallel and Distributed Processing with Applications*, pages 99–112, Washington, DC, USA, 2008. IEEE Computer Society.

[335] Abdallah Deeb I. Al Zain. *Implementing High-level Parallelism on Computational Grids.* PhD thesis, School of Mathematical and Computer Sciences, Heriot-Watt University, March 2006.

[336] Ying Zhang and Peter Boncz. XRPC: interoperable and efficient distributed XQuery. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 99–110. VLDB Endowment, 2007.

[337] Gang Zhou. Dynamic Dataflow Modeling in Ptolemy II. Technical Memorandum UCB/ERL M05/2, University of California, Berkeley, CA 94720, December 2004.