A Distributed Virtual Machine for Parallel Graph Reduction

Peter M. Kelly, Paul D. Coddington, and Andrew L. Wendelborn
School of Computer Science
University of Adelaide
South Australia 5005, Australia
{pmk,paulc,andrew}@cs.adelaide.edu.au

Abstract

We present the architecture of nreduce*, a distributed virtual machine which uses parallel graph reduction to run programs across a set of computers. It executes code written in a simple functional language which supports lazy evaluation and automatic parallelisation. The execution engine abstracts away details of parallelism and distribution, and uses JIT compilation to produce efficient code.*

This work is part of a broader project to provide a programming environment for developing distributed applications which hides low-level details from the application developer. The language we have designed plays the role of an intermediate form into which existing functional languages can be transformed. The runtime system demonstrates how distributed execution can be implemented directly within a virtual machine, instead of a separate piece of middleware that coordinates the execution of external programs.

1 Introduction

Distributed computing systems designed for executing a large parallel computation among a collection of different computers have become common in recent years. They permit computationally demanding problems to be solved relatively quickly, often reducing the amount of time required from years to weeks. The participating machines may either be owned by regular users volunteering their idle CPU time for a good cause, or owned by a single organisation that uses idle workstations to do its compute intensive processing. This is a cheap alternative to a cluster, since it uses resources that are already there but are under-utilized when users are not physically present. The same approach can also be used on shared memory systems to distribute work among processors.

Most such systems hand out *independent* jobs to different machines, possibly with some level of coordination used to manage the dependencies between different jobs. However, many of the opportunities for parallelism can actually reside *within* a job. The programs are run directly on top of the operating system, or within a virtual machine such as Java or .NET. These do not permit different threads to be easily distributed to remote machines, at least not using the mechanisms that the distributed computing middleware provides for distributing jobs. Finer grained control over parallelism can be achieved using message passing libraries such as MPI, but at the cost of dealing with task interaction and data distribution manually, which is more challenging and error-prone.

We have taken an alternative approach to parallel programming for such environments by developing a programming language and runtime environment specifically designed to integrate scheduling with the execution of the actual program. Rather than permitting scheduling only at the level of coarse-grained, independent jobs, our system supports complex dependencies and finer grained distribution of work, right down to the level of individual function calls. This opens opportunities for taking advantage of additional parallelism, while providing us with the ability to experiment with a range of different granularities in a way that is not normally provided by middleware.

Our system is based on a combination of ideas from both the parallel functional programming and distributed computing research communities. It is a virtual machine that executes applications written according to the functional programming model, and automatically extracts parallelism and manages the distribution of work between different computers. The side effect free nature of functional programming facilitates automatic parallelisation and provides us with an effective means of hiding low-level distribution and task interaction details from the programmer. The research we are conducting is intended to bridge the gap between the largely separate worlds of coarse grained job scheduling and fine grained parallel programming and investigate the benefits that can occur when ideas from these areas are integrated.

2 Related work

Condor [21], XGrid [7], and Sun Grid Engine [5] provide the ability to distribute jobs among a collection of cluster nodes or user workstations located on a local or wide-area network. Each job is a separate program instance, which the middleware runs, collects the output, and makes the results available to the user. Other projects such as OptimalGrid [11] and Alchemi [17] provide similar facilities for Java and .NET-based programs, dividing the tasks into different objects instead of executable programs. In some cases it is possible to specify dependencies between jobs [8, 18], in a programming style often referred to *dataflow* or *workflow*.

The specification of dependencies between jobs is essentially a type of high-level programming model where constructs are provided for specifying tasks and dependencies. This model is similar to *graph reduction*, which has been used at a much finer grained level to implement many functional programming languages. Two such implementations, the (v,G)-machine [3], and GUM [22], have inspired our own implementation, particularly the latter which has recently been extended to utilise machines in different geographical locations [23]. A detailed study of load balancing approaches for distributed memory graph reduction is given in [16], and other work has shown success in implementing the concept on shared memory systems [6].

While the (v,G)-machine runs only on shared memory systems, both our system and GUM support distributed memory execution, where the heap is partitioned between different computers. We utilise a similar model of parallel execution to that of GUM, the main difference being the way in which communication is handled. GUM does not implement inter-node communication itself but instead uses either PVM [4] or MPI [19], both of which rely on a central point of control and don't interact well with sockets. We instead use our own proprietary message passing layer based on a peer-to-peer model, described in Section 4.2. In our communications model, there is no "master" node that can bring the whole system down if it fails. It also permits our system to be deployed as a stand-alone package, which does not require the installation of extra libraries which are not normally present in workstation environments.

While it is possible to write message passing programs directly using PVM or MPI, these are both very low-level programming models that require a lot of expertise. Higher-level languages that abstract away complex details have the potential to make parallel programming easier. As with other distributed memory models, our approach is also applicable to shared memory systems such as multi-core processors.

Our virtual machine and intermediate language are part of a broader project to implement a distributed, parallel version of XSLT [12]. The use of graph reduction came about as an evolution of earlier work on compiling XSLT into dataflow graphs [13], while our stream-based implementation of network connections is intended for use as an underlying mechanism for accessing web services [14].

3 Graph reduction

The dataflow model used by job scheduling systems uses *directed acyclic graphs* (DAGs), where each vertex represents a job, and the edges indicate dependencies between them. A job can only begin execution once all of the other jobs it depends on have completed, and the output files produced by those jobs are used as the subsequent input data.

Graph reduction also represents the program as a graph, but in a different manner. Each vertex represents either a function reference, a call to a function, or a computed value. A "job" in the DAG model corresponds to a call to a function with one or more expressions, where each expression may be any of the three types of vertices. Graph reduction is a popular implementation technique for functional languages because of its support for lazy evaluation, and the existence of techniques for compiling such programs into efficient sequential code.

We use a modified version of graph reduction based on the (v,G)-machine [3], which uses a collection of *frames*, organised in a heap. These are similar in nature to the stack frames used when executing imperative languages, except that they are organised as a graph, rather than a stack. Within an individual frame, the instructions corresponding to the relevant code are executed sequentially. Parallelism is achieved by detecting cases where the results of more than one function call are needed by an expression, and *sparking* frames for each of the calls, making them candidates for execution. Sparked frames can then be distributed to worker machines in the same way that jobs are in a typical task farming system.

The graph is stored in a distributed manner across each of the machines, and data values are exchanged between machines as necessary using request/response messages. Distributed garbage collection is performed at regular intervals to clean up parts of the graph that are no longer referenced, after having been consumed by functions that have finished executing.

Our work thus takes the concept of DAG-based scheduling and approaches the problem from the perspective of implementing a functional programming language. The unit of execution is much more fine grained; instead of jobs we deal with function calls, and the graph maintained at runtime includes the actual output data that is computed and passed as input to subsequent operations.

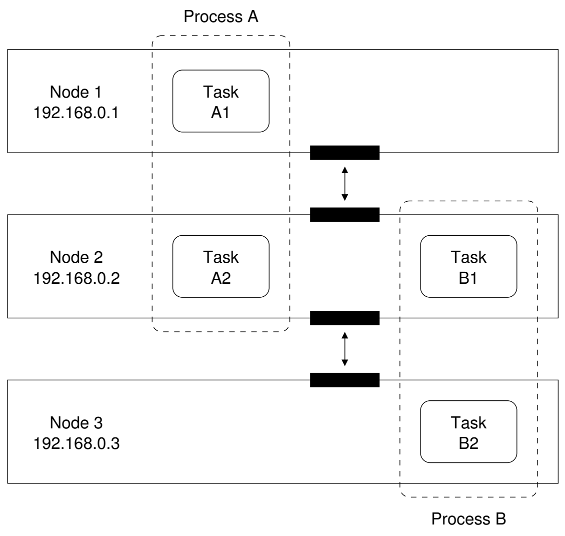

Figure 1. A set of nodes with running processes

4 Distribution model

4.1 Nodes and tasks

A separate instance of the virtual machine runs on each computer, and is referred to as a *node*, each of which maintains connections to several others. The organisation is based on a peer-to-peer model, avoiding reliance on a central point of control. We use the Chord protocol [20], since the number of connections each node has open is logarithmic in the size of the network, and joins and departures are handled robustly. Nodes can join or leave the system at any time, based on user control or automatic launching of the virtual machine on participating hosts.

A program that is running on a set of nreduce nodes is referred to as a *process*. Each process consists of a set of *tasks*, each residing on a different node. In general, only a subset of all available nodes will take part in any given process. A node may be running tasks from several different processes. An example deployment is shown in Figure 1, containing three nodes with two running processes, each of which has two tasks. The black edges in the figure represent socket connections between the nodes. On multi-processor machines, parallelism can also be obtained within a single VM instance since each task is run in a separate thread.

A user launches a process by using a command-line client, which takes arguments specifying the name of the source file, the address of an existing node, and the number of machines to be used for execution. The client program first compiles the source code, reporting any errors it finds to the user. If compilation succeeds, it connects to the specified node and performs a search on the network for available hosts that can be used for execution (details of which are outside the scope of this paper). Once the node set has been determined, the bytecode is sent to each node; the nodes then perform just-in-time (JIT) compilation of the bytecode and begin execution of the task.

4.2 Messaging

Communication between tasks on different nodes occurs using message passing. While PVM and MPI were originally considered for use, neither met all of our requirements. For reliability purposes we wanted a peer-to-peer mechanism that did not rely on a central point of control, so that no failure of an individual node or group of nodes would bring the whole system down. To support distributed heap management, tasks also need a mechanism by which they can be interrupted whenever a message arrives, instead of paying the cost of explicitly polling for message availability while executing compute intensive code. No message passing library we are aware of provides these features - particularly the latter, since it needs tight integration with the JIT compiler to ensure interruptions only occur at safe points. Using an integrated messaging layer also makes deployment easier, as it is simpler to install a single statically-linked executable on a set of desktop workstations than to rely on a correctly configured MPI installation already being present.

Each task runs within a separate thread, and has an *endpoint* object associated with it that is used to send and receive messages. An endpoint is identified by a tuple $\{ip, port, localid\}$, where *ip* and *port* refer to the IP address and port number used by the node, and *localid* is a locally unique identifier assigned to the endpoint. Endpoints are also associated with other threads that run within the VM, such as the manager thread used to coordinate process creation. Each message has a header which records its source and destination, a tag indicating the type of message, and a variable-sized payload. A task only receives notification of message arrival once the entire message has been received by the node.

Each node runs a dedicated I/O thread to handle communications. It maintains socket connections to other nodes, and uses a *select()* loop to wait for data to be received over the network, or in the case where there are outgoing messages waiting to be sent, for sockets to become writable. When data is received over a connection, it adds it to a read buffer, and as soon as that buffer is found to contain a complete message, it delivers the message to the appropriate endpoint. Tasks get notified of message arrival by the I/O thread sending a signal to the task's thread, which causes it to jump out into a message handling function. The other threads explicitly issue a *receive* operation and block while they are waiting for the next message.

The messaging model is send-and-forget. When one endpoint wants to send a message to another, it invokes a *send* operation, specifying the destination endpoint id, the tag, and the binary data to be sent. This is handed to the I/O thread which places it in an outgoing queue for the appropriate connection and handles the actual data transfer. From this point, the sending endpoint gets no notification of whether or not the message was successfully received. A *linking* mechanism similar to that of Erlang [1] permits a thread to be notified when one of its peers fails.

5 Program execution model

5.1 Input language

Programs run by nreduce are written in a language we refer to as *extended lambda calculus* (ELC). This is a very simple programming language based on the well-known lambda calculus model shared by other functional languages such as Haskell [10]. Basic features such as arithmetic expressions, conditional tests, and list manipulation functions are supported on top of the underlying model, all of which follow standard functional semantics. Since our research is concerned with the implementation of the programming model, we will not go into details of the syntax or other specifics here, with the exception of some additional language features described in Section 6. Our implementation techniques can be applied to other functional languages that share the same basic model.

While it is possible to write code directly in ELC, it is primarily intended as an intermediate form into which other languages such as XSLT can be compiled. However, such compilation and usage is outside the scope of this paper; we instead focus here on the approach used for executing and compiling ELC programs.

5.2 Frame management

Within a process, each task is responsible for executing some portion of the work available to be done, and storing part of the graph. Each task maintains its own heap, which contains a series of *cells*, which are the vertices of the graph. References between cells in the same heap are via a standard pointer to the memory address of the other cell, while references to cells in other heaps go through a separate indirection layer,

The basic unit of execution is a *frame*, which represents a call to a function with a particular set of actual parameters. The creation of a frame is a separate action to its invocation. A frame that has been created but has not yet started executing corresponds to a "suspended evaluation" or "thunk" common in many functional languages. It is a value that can be passed around in memory as arguments to other functions, or returned as the result of a function. When a frame begins execution, the virtual machine executes the instructions at the code address associated with the frame. In some cases execution will begin immediately after a frame has been created; in others it may be created and either evaluated later, or possibly even never evaluated if it turns out its result is not needed.

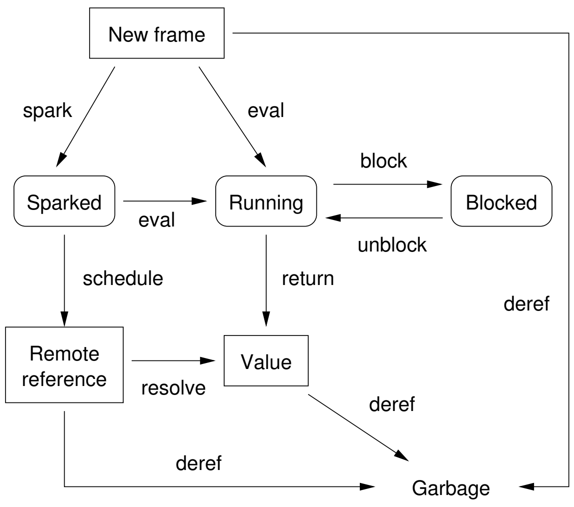

Figure 2. Life cycle of a frame

The management and distribution of frames is based on the approach of [22]. Each task maintains three sets of frames: *sparked*, *runnable*, and *blocked*. Sparked frames are those that the program has determined it will definitely need to execute at some point in the future, but whose results are not yet needed. Runnable frames are those which have begun execution and have work that they need to do now. Whenever the runnable set is non-empty, the processor is always executing one of them. Blocked frames are those that have begun execution, but are waiting either for another frame to return, or for requested data to become available from a network connection. The runnable and blocked sets are analogous to those that an operating system kernel maintains for processes/threads.

The state transitions that a frame may go through are shown in Figure 2. When a frame is initially created, it is not in any of the sets. If it is sparked, it will be put into the sparked set, and at some later point in time will be executed locally by the task, or sent to a remote task for execution. When another frame needs the result of the frame, it will begin evaluation of it, and place it in the runnable set. The frame may be blocked and unblocked several times during its execution, if it calls other frames or makes requests over the network. Finally it will return, and the heap cell associated with it will be updated with the result value.

If a frame is assigned to another machine for execution, then the cell in the local heap which referred to that frame becomes a remote reference. If the value of this reference is needed by another running frame, a request will be sent to the remote machine for the value. The remote machine will reply with the value once the frame has finished executing, and the cell in the requesting task's heap will be changed to hold the value.

If a frame is not in the sparked, runnable, or blocked sets, it may become dereferenced. This means that there are no longer any references to it from other frames or cells and it is considered garbage. This is not possible while it is in one of the three sets, because for it to have been put in those sets there must be at least one other place from which it is referenced, which is waiting on the result of the frame. When it does become dereferenced, either before or after it has been executed, it will become a candidate for garbage collection, and will be deleted when the next collection is performed.

5.3 Work distribution

When a process starts, it consists of a set of idle tasks, each running on a different processor. The number of tasks that takes part in a process is specified by the user when the process is created, and each accepts responsibility for executing some portion of the program. All tasks except one start out with no runnable frames; the lowest-numbered task initially contains a single frame correspdonding to a call to the main function of the program.

As execution proceeds, additional frames will be created. Those whose values are needed immediately will be evaluated by the local task, while others that are found to be needed in the future are sparked. This sparking occurs based on strictness annotations in the code, derived during the compilation process described in Section 5.4.

To cater for nodes with different processor speeds and the difficulty of predicting ahead of time how long a function call or expression will take to evaluate, we use a dynamic load distribution mechanism. An idle task (one with no running frames) sends out a work request to a randomly chosen task. If that task has frames available in its sparked set, it migrates some of them to the requester; otherwise, the work request is forwarded to another task. A time-to-live field associated with the request is decremented each time so that the request will eventually disappear. If an idle task does not receive a responses after a certain time period, it sends out another request.

Upon receipt of one or more sparked frames, a task will place them in its running set and begin executing the first one. The relevant heap cells on the originating task will become remote references that point to the frames at their new location. If an attempt is made to evaluate one of these references, a request will be sent to the task running the frame, and the response will be sent back once the frame has completed. Such a request causes only the frame that made it to block; other frames which do not depend on the requested value may continue execution; this is effectively a form of latency hiding.

5.4 Compilation

ELC programs are compiled in a similar manner to that described in [9], using the following steps:

1. The source file and imported module files are parsed into a syntax tree.
2. Variable references are checked, and resolved to point to the appropriate identifiers.
3. Lambda lifting replaces all lambda abstractions with supercombinators (top-level functions).
4. Optimisation is done by performing a series of transformations on the syntax tree. These include in-lining of simple function calls, and modifying certain constructs to reduce the number of frames that must be allocated due to lazy evaluation.
5. Strictness analysis is performed to determine which expressions each function definitely needs to evaluate, and can thus be sparked when the function is called. This is only done when it is certain their results will be needed, to avoid sparks causing unnecessary work being done, which could possibly lead to infinite loops.
6. The resulting modified/annotated syntax tree is compiled into platform-independent bytecode.
7. The bytecode is compiled into native code for the target machine. The code generator currently produces x86 code.

Step 7 is separate from the main compilation process, and is optional. Two variants of the execution environment are provided: an interpreter, which operates directly on the bytecode, and a native code engine, which runs the compiled code. When a client launches a process, it submits the bytecode to each node, which can then choose to either use the interpreter or JIT engine.

5.5 Execution

The interpreter implements an abstract machine conceptually similar to the G-machine [2]. The instruction set is specifically tailored toward graph reduction, and includes operations for creating and evaluating frames, executing built-in functions, as well as the usual arithmetic and control instructions. In addition to executing the program's bytecode instructions, the interpreter interacts with other parts of the runtime environment for actions such as handling messages from other tasks and performing garbage collection.

Native code execution is based on the same abstract machine concept. Before execution begins, the code generator performs JIT compilation to translate the bytecode instructions into machine code. This avoids the overhead of the runtime tests of the interpreter such as the top-level switch statement, as well as other operations that can be either skipped or done more efficiently in the generated code. Also, where the interpreter checks for exceptional situations like a message becoming available on each instruction, the native code engine relies on signal handlers to deal with these, avoiding the cost of constantly checking for these cases. The code generator is designed in such a way that it can call back to the interpreter for certain instructions, while executing others directly. This simplifies additions or changes to the instruction set and aids debugging.

6 Additional capabilities

The main concepts of ELC are based on the well-known lambda calculus model used by other functional languages, so will not be discussed in detail here. However, there are a few design choices we have made which differ in some respects to other languages and warrant further discussion.

6.1 Automatic parallelisation

Parallel functional languages often require explicit annotations from the programmer to indicate which expressions should be executed in parallel, such as Haskell's *par* combinator [22]. This is due to the overhead associated with sparking expressions, which can be minimised by only doing so where specifically requested by the programmer. We have elected to instead add these annotations automatically during strictness analysis, to relieve the programmer of this responsibility, and cater for high-level languages which do not include support for such annotations. To compensate for the fact that the compiler adds more annotations than would usually be specified by a programmer, we only spark expressions involving function calls, and have optimised our implementation to minimise the overhead of sparks when they do occur.

If a frame is already on the heap, then all our implementation needs to do to spark it is to change its *state* field to "sparked". This cost is small, because it only requires the cell type to be checked and a single field to be updated. Rather than maintaining the set of sparked frames in a data structure such as a linked list, any time the sparked set needs to be accessed, a heap traversal is performed, and the *state* field of each frame is examined. While this traversal is more expensive than looking at a small linked list of frames, the overall impact is less. This is because modifying the sparked set happens regularly, and accessing it only happens on the relatively infrequent occasion that the task receives a work request from another machine.

6.2 Data representation

In order to keep the programming model simple, ELC only supports three data types: numbers, cons cells, and the special value *nil*. Data structures such as lists can be built up of multiple cons cells in the standard way. Strings are simply lists of numbers, each of which is a character code. Binary data read from files and network connections is also represented in the same manner.

While strings and binary data appear to the program as lists, they are actually stored internally as arrays [15]. This permits contiguous blocks of memory to be used for the contents, avoiding the overheads of large linked lists, and enabling data read from a socket to be directly copied into the array. The built-in list operations such as *head*, *tail*, and *length* operate on arrays and lists in such a manner to hide the underlying storage mechanism.

6.3 Streams

When an ELC program creates a network connection, a list abstraction is created for input and output. The data read from the connection is exposed to the program as a list, and lazy evaluation is used to control reading of the data. As the program traverses the list, more data is read from the connection. When a frame attempts to evaluate part of the list that has not yet been read, it will block until the data is received from the other side. When the data arrives, the frame will be woken up and given the requested list elements. If only an initial portion of the list is ever accessed by the program, later parts which become dereferenced will be deleted by the garbage collector, which will cause the connection to be closed.

Output works by invoking a user supplied function for each new connection, and interpreting the result of that function as a list of bytes to be written to the connection. The function may either return a string value directly, in which case the data would be written immediately, or it can use lazy evaluation to produce a longer data stream. In the latter case, the runtime environment will traverse the resulting list and write out the data as it becomes available. If the operating system's buffer becomes full because the other side has not yet read the data, the traversal will block. When the buffer contains some available space, the socket will become writable, and the traversal will unblock and cause more data to be written.

Because this approach to I/O utilises the runnable and blocked frame sets, it enables ELC programs to take advantage of the parallel execution semantics even on a single processor. Multiple expressions involving different network connections can be active at the same time, and when one or more of them are blocked while waiting for data, the others can continue. This is particularly useful for programs which interact with multiple external network services, or act as a service themselves and support multiple client connections.

7 Current status

A prototype implementation of the system has been under development for close to 18 months, and consists of around 22,000 lines of C code. It runs on both Linux and Mac OS X. The underlying operating system is completely hidden from ELC programs, and nodes on different platforms can cooperate together in the execution of a process. We have tested parallel execution on a cluster of 60 nodes, and developed a number of sample applications such as a web server, Mandelbrot generator, and XML parser, to verify that the VM is able to run reliably and produce correct results.

Extensive testing has also been done on our implementation of the Chord protocol. Simulations of up to 256 nodes have been carried out using multiple threads on a single machine, as well as real deployments on the test cluster. In both cases, the networking infrastructure is able to tolerate multiple simultaneous node failures and repair itself quickly.

Now that our implementation is working reliably, we are addressing the issues of latency that typically affect communication intensive programs. These efforts include improving dynamic scheduling, and minimising the number of messages required for transfer of graph segments. Although work such as [16] and [22] has looked at these issues, further research is needed in the context of peer-to-peer environments. Additional optimisations in this area are needed to make our system fast enough for real-world usage.

8 Future work

Fault tolerance is not currently provided within the context of a process. A node failure will cause all processes with tasks on that node to be terminated, since parts of the heap stored on that node may be needed. The side effect free nature of our programming model permits the possibility of recovering lost data through re-computation, provided that the necessary information can be obtained from other nodes. This would be an interesting extension of our work, and would further demonstrate ways in which a side effect free language can be beneficial.

For user workstations, background utilisation of a running VM can be problematic. Cycle stealing systems generally provide a mechanism to suspend or migrate computation elsewhere if a user becomes active at the machine. Ideally, our system could support such a mechanism through integration with a screen saver-like mechanism to detect periods of inactivity.

Aside from this additional functionality, the most important work remaining relates to performance improvement. This includes work distribution mechanisms, distributed garbage collection, and code optimisation. While our focus has mostly been on new research ideas, much existing knowledge on compiler and virtual machine technology exists that could be applied to our system.

9 Conclusion

Our virtual machine abstracts away low-level details of parallelism and distribution of functional programs, removing the need for the programmer to explicitly deal with these issues. Additionally, the stream-based model of network I/O enables many network connections to be handled concurrently by a program in a transparent manner. These features provide a simple, high-level programming model that allows programs to easily benefit from concurrency and parallelism, with less complexity than many traditional programming environments. Numerous optimisations have been made to to the compiler and runtime system with the goal of improving efficiency.

Evaluation and results have not been provided in this paper due to space restrictions and the need for further optimisations to be made to the VM before good performance can be obtained. Our intention here has been to present the features and architecture of our system and to give an overview of the implementation. Future work will lead to a detailed evaluation of system performance and address issues relating to fault tolerance and the impact of network latency.

We believe that distributed computing systems based on coarse grained jobs can benefit from increased granularity and the use of a functional programming model. Our work combines these two ideas in an attempt to explore how practical finer grained work distribution is in various scenarios such as peer-to-peer networks of workstations.

References

[1] J. Armstrong. The development of Erlang. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 196–203, New York, NY, USA, 1997. ACM Press.

[2] L. 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.

[3] L. Augustsson and T. 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.

[4] A. Beguelin, J. J. Dongarra, A. Geist, S. Otto, and J. Walpole. PVM: Experiences, current status and future direction. In *Proceedings of Supercomputing '93*, pages 765–766, Portland, Oregon, November 1993.

[5] W. 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.

[6] T. Harris, S. Marlow, and S. 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.

[7] B. Hughes. Building computational grids with Apple's Xgrid middleware. In *ACSW Frontiers '06: Proceedings of the 2006 Australasian workshops on Grid computing and e-research*, pages 47–54, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[8] C. Jin, Z. Zhang, L. Stein, and R. Buyya. A dataflow system for unreliable computing environments. Technical Report GRIDS-TR-2006-18, Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia, November 2006.

[9] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[10] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, January 2003.

[11] J. H. Kaufman, T. J. Lehman, G. Deen, and J. Thomas. OptimalGrid – autonomic computing on the grid. *IBM developerWorks*, June 2003.

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

[13] P. M. Kelly, P. D. Coddington, and A. 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.

[14] P. M. Kelly, P. D. Coddington, and A. 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.

[15] P. M. Kelly, P. D. Coddington, and A. L. Wendelborn. Efficient list representation for lazy functional languages. Technical report, DHPC Group, Computer Science Department, Adelaide University, 2007. To Appear.

[16] H-W. Loidl. Load balancing in a parallel graph reducer. *Trends in Functional Programming*, 3:63–74, 2002.

[17] A. Luther, R. Buyya, R. Ranjan, and S. Venugopal. Peer-to-peer grid computing and a .NET-based alchemi framework. In *High Performance Computing: Paradigm and Infrastructure*. Wiley Press, Fall 2004.

[18] G. Malewicz, I. Foster, A. L. Rosenberg, and M. Wilde. A tool for prioritizing DAGMan jobs and its evaluation. *Journal of Grid Computing*, 5(2):197–212, June 2007.

[19] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, 1995.

[20] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *ACM SIGCOMM 2001*, pages 149–160, San Deigo, CA, August 2001.

[21] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 2004.

[22] 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.

[23] 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.