Research Proposal:
A Unified Approach to Scheduling in Grid Environments

7th October 2004

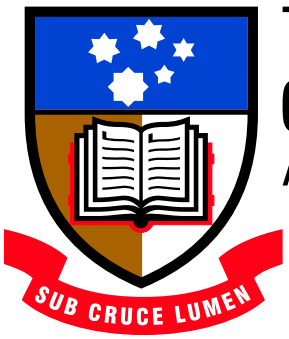
Peter Kelly, B.Info.Tech., B.Comp.Sci (Hons)

School of Computer Science

The University of Adelaide,

South Australia

Supervisors: Dr. Paul Coddington and Dr. Andrew Wendelborn



THE UNIVERSITY
OF ADELAIDE
AUSTRALIA

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Abstract

Grid computing systems provide mechanisms to utilise a wide range of heterogeneous, distributed resources for compute- and data-intensive applications. In order to obtain good performance, it is necessary to select resources for use in such a manner that minimises the computation and communication time. The process of scheduling, or deciding which resources to use, has been explored for many different types of grids and applications. Three types of scheduling are used: *computation scheduling*, involving selecting hosts to execute a program, *data scheduling*, determining the placement of data for efficient access, and *service scheduling*, the selection of a remote host on which to access a particular service. Existing schedulers generally focus on only one of these types, and do not consider the interaction between them. We propose a model for scheduling grid applications based on the problem of assigning the *schedulable entities* of a grid application to resources. These entities represent the computation, data and service components of an application. Information about these entities and the relationships between them will be taken into account when making placement decisions. Examples of such relationships include the communication between tasks, and data dependencies between tasks and files. Our investigation will include the implementation of this model in several different grid middleware systems.

Contents

1 Introduction

1.1 Grid Computing

The term *grid computing* refers to ways in which distributed, heterogeneous sets of computing resources can be organised together to provide facilities in such a coordinated, consistent manner. A grid may consist of resources owned by a number of different organisations, within which sharing arrangements have been established. Users can access resources on the grid to perform various tasks related to their work, thus gaining access to additional storage, processing power and other facilities that would not otherwise be available to them.

Many different types of grid computing systems have been developed over the years. These vary widely in the specific platforms and standards they use, as well as in their architecture and targeted usage scenarios. There are even a range of definitions of what exactly constitutes a grid; different projects and organisations work off different interpretations of the term. However, the general overall concept is fairly similar in all of these cases - that of harnessing together a range of different resources for usage by applications.

There are other areas which are related in some respects to grid computing, but differ from it. *Cluster computing* [20] refers to parallel computers built from a set of machines of the same type, residing in the same location and under the control of a central authority. *Distributed computing* [73] is a very general term relating to a range of different technologies, from remote procedure call (RPC) mechanisms to distributed object systems and others where computation occurs across a number of different machines. *Client-server computing* [62] is an architecture consisting of a single server providing computational and storage resources which is accessed by multiple clients.

The concept of Grid computing, while not always known by that name, has a long history dating back to the earliest days of the Internet, predating even TCP/IP itself. A 1970 experiment demonstrated the coordinated use of PDP-10s and PDP-1s located at Harvard and MIT to run an aircraft carrier simulation program distributed across three geographically separated nodes [103]. One of the first programs to utilise unused cycles of a series of workstations was called *Creeper*, a worm which replicated itself from machine to machine via ARPANET [55]. Early work on load balancing algorithms for running applications across distributed systems [74] lead to the creation of the Condor project [115] in the early 1980's, one of the first such systems capable of scheduling and assigning jobs to a series of workstations. Many of the concepts prevalent in the field of grid computing, including services, remote process invocation, metascheduling, resource management, naming, distributed parallel programs, networks of workstations, and load balancing were already well established by the mid 80's [110], and have been implemented in a wide range of commercial, open source, and research systems over the past twenty years. These are forming the basis of much of the grid architecture being built today; the widely distributed, heterogeneous nature of grids however introduces additional implementation challenges, and this past work is being built upon to create additional functionality and systems that will scale to the demands of today's grids.

Many grids have been developed over the years, supporting a wide range of platforms, application types, and usage models. In recent years, efforts have been made to standardise grid technologies to enable increased interoperability, with the creation of the Global Grid Forum [23]. This group of researchers and organisations has made significant progress in capturing "best practices" from a range of different projects and is in the process of defining standards which, once widely implemented across the majority of mainstream grid software, will lead toward easier creation of large scale grids and sharing of resources between parties. Some of the outcomes of these efforts include the Globus Toolkit [45] and the OGSA (Open Grid Services Architecture) [47]. While significant progress has been made, much work still remains to be done before the vision of a world-wide grid capable of efficiently and securely supporting all types of applications and user needs is realised.

One significant area which needs further work toward this is that of making the most efficient use of the resources that are available. While large numbers of powerful computers are available and capable of being connected to a grid, the problem of assigning data and computation to them in such a way to optimise performance is a challenging one. This is particularly true for applications that require the use of multiple resources, such as parallel programs or those that access data or services on other computers that are not available locally. Many grid systems now include a *metascheduler*, which selects resources for use on an automated basis, using information about the applications and properties of the resources that are available. While these have made it easier for users to run their applications in a grid environment, much work still remains to be done before these provide seamless and efficient ways of utilising resources by freeing the user from the underlying details of the grid. This project aims to make some progress in this direction, and provide insights on to how this process can be made more effective for certain classes of applications and grids.

1.2 Project Overview

The research proposed for this project aims to develop an architecture and algorithms for scheduling different components of an application in order to optimise performance in a grid environment. The three types of components covered are computation, data, and services, and are described further in Section 2. Due to the significant differences in the ways each of these component types is supported in different programming systems, our research will focus primarily on the generic aspects of the problem, and also demonstrate its implementation in several different systems. Our intention is to implement the scheduling algorithms in such a manner that the implementation will support a number of different types of grid software.

The core ideas relating to our model have grown from an analysis of existing technologies for grid and parallel computing, which are further elaborated on in this proposal. Much work has been done in the field to date, providing a solid base on which to pursue our research. We will be making extensive use of ideas and technologies that have been developed previously, in order to leverage them for use in our work, and to ensure that the approaches and algorithms developed or investigated within the context of this project have utility for widely deployed systems, both now and in the future.

The main novel idea behind our work is that we consider the interaction between different components of an application when making scheduling decisions. Examples of these interactions include the degree of communication between tasks, and the data access patterns of an application. Previous work in the field has mainly concentrated on scheduling each task or other type of entity separately without consideration for how it interacts with the rest of an application, or just scheduling computation by itself without regards for the communication costs associated with placement decisions. We propose a unified approach which considers all of these things together, to provide a more rounded approach to achieving optimal performance.

Our investigation will begin by looking at a particular grid programming environment that already supports many of the features that are needed for research into scheduling strategies. The software used will be PAGIS [124], a system based on process networks which integrates with Globus to support execution of programs in a grid environment. Currently the mechanisms used for scheduling computation in this system are very simplistic, and we will begin by implementing more advanced scheduling strategies. We will also add support for data access and grid service integration, extending the scheduling support to include these aspects of programs as well.

Once the concepts have been implemented in PAGIS and the issues surrounding scheduling of grid applications are more deeply understood at a practical level, we will we will extend our research to other existing programming systems that are in use on various grids. An area of particular interest is the emerging OGSA model, and other standardisation work being done by the Global Grid Forum, although other systems will be looked at as well. The implementation of our ideas into existing systems will serve to demonstrate the widespread applicability of our approach, and to show how the theoretical aspects of our research are directly relevant to real-world problems.

2 Background

A large amount of work has been done in the field of distributed high performance computing over the past three decades. This has been in a number of areas, ranging from high-end super computers, to clusters, client-server systems, parallel programming systems and others. Recently, the term "grid computing" has become popular to describe some of ways in which these technologies have been combined to provide easier ways to make use of computing resources spread across the world. This section gives an overview of a number of important parts of the field, and describes some of the past work that the research proposed in Section 3 will build upon.

2.1 Types of Grids

The term *grid computing* has been used to describe many things, and before embarking on a discussion of the field, it is important to define what is actually meant by this term. In this proposal, it is used to refer a large, geographically distributed collection of heterogeneous computers residing in different administrative domains, cooperating with each other through middleware software to enable usage of the collection of machines in an easier manner. This is a very general description, and can apply to many types of configurations. To further clarify this, and to apply it to the major areas of the field, we draw a distinction between what we term *heavyweight* grids and *lightweight* grids.

Heavyweight grids are built out of big, expensive supercomputers or clusters owned by academic or research institutions and large corporations. Each of these computers are generally bought by the organisation for the purposes of running computationally demanding software used for discipline-specific purposes in science, engineering, or business. The machines contain specialised mechanisms for submitting and monitoring jobs, and often dedicate processors to processes, rather than multitasking jobs on a processor. Administering such machines is a non-trivial task, and requires expertise in understanding many details of how the system operates, and the skills to carry out complex software installation and configuration procedures. Full-time staff are often hired specifically to administer these machines. Such systems are made available to a grid by their owners according to sharing arrangements with other institutions, such as members of an international research collaboration or industrial consortium, often termed a *virtual organisation*. Because of the large investment of money, time, and personnel into these systems, owners understandably want a say in how they are used by other members of the grid, and thus define specific policies which state who has access to the resources, how and when they are used, and other criteria.

Lightweight grids consist primarily of desktop machines sitting on users' desks, often termed a *network of workstations* [6]. These may reside solely within the one organisation, span across multiple organisations, or consist of an Internet-wide collection of computers both in people's homes and offices. These grids are designed around the concept of *cycle-stealing*, that is, making use of idle processors in these workstations when they are not being used for their normal purposes. The policies imposed on each individual host are generally much simpler than those of high-end supercomputers - often they simply state that the computer can be used for any processing desired by the grid, as long as it doesn't run while the user is at the machine. In many cases, users neither know or care what the grid software is doing while they are away from their machines, as long as the processing is done securely and their data is not at risk. Due to the lack of system administration skills possessed by most people, ease of installation is a critical factor in the success of any grid solution targeted to an Internet-scale deployment of such systems.

The way in which jobs are launched differs significantly between supercomputers and workstations. A supercomputer contains a local job submission system which places jobs in a queue and then executes them when it deems appropriate. While a job may start executing immediately, it could potentially be postponed to some point in the future, if it needs exclusive access to processors and the required number of processors is busy running other jobs. The scheduling strategies used by supercomputers are discussed further in Section 2.5.1. On the other hand, grid systems designed for cycle stealing on workstations typically do not maintain local queues on each machine, so while a job may still need to wait in the global scheduler's queue, once it has been dispatched to a machine it is guaranteed to start without additional delays. When running a parallel application on a lightweight grid, it is therefore not usually necessary to take into account the differences between local job scheduling strategies between hosts.

The distinction between heavyweight and lightweight grids, and the software designed for them, is not always clear. Most systems designed for networks of workstations can also be deployed in a cluster environment, and middleware aimed at connecting supercomputers together can, at least theoretically, support large collections of workstations. However, traditional cycle-stealing systems often ignore the autonomy and complex local job scheduling policies often present on supercomputers, and software designed for heavyweight grids is not always as easy to deploy as systems designed for desktop usage.

An example of a heavyweight grid is the systems targeted by the Globus toolkit. A system administrator installs Globus on a parallel computer or cluster, and configures it according to their needs. They must set up the job manager to work with their local scheduler, and register the host with the directory service, which often runs on a separate machine. For security purposes, a host certificate, which uniquely identifies the machine and enables clients to connect to it, must be obtained from some central authority. Additionally, a "gridmap" file must be created which specifies a mapping of user certificates to entries in the local user database. This configuration is very machine-specific and allows for a high degree of control over how the machine integrates with the grid and who is allowed to use it. However, the complex installation process is generally inappropriate for mass deployment on individual workstations, because of the amount of work involved and the level of expertise it requires [29].

Numerous systems designed for lightweight grids have been developed in recent years. Entropia [28] enables the use of windows-based desktop machines in an organisation for running grid applications submitted to a central job manager. It provides a security mechanism to protect the local machine from accidental or malicious damage by grid applications using a sand-boxing technique involving the modification of native win32 binaries to prevent grid applications from performing unauthorised operations or accessing user data. It also contains mechanisms to ensure the integrity of the application and its associated data files. The POPCORN [80] and SPAWN [121] projects allow workstations to participate in parallel computations and use an economic market system for trading CPU time. ParaWeb [19] and XtremWeb [51] are similar systems capable of executing parallel Java programs across a number of nodes, and Alchemi [76] provides essentially the same functionality using the .NET framework. A number of application-specific grids have been built in which large numbers of users, sometimes in the millions, install client software designed to solve a particular problem. Prominent examples include the SETI@Home project [66], which utilises the idle cycles of users' machines to search astronomical data for signs radio signals indicating the existence extraterrestrial intelligence, distributed.net's efforts to crack the RC5 encryption algorithm [43], and Folding@home [68], which utilises client machines to simulate protein folding experiments used for medical research. Condor [115] provides a mechanism for accepting job requests from clients and assigning them to machines based on a load balancing strategy. These machines can be running any platform that supports the Condor software, and are selected for job execution based on their operating system, and other parameters that are specified by the user. Job requests are matched to machines using a mechanism called ClassAds. Numerous execution environments are supported, including native binaries, Java programs, and parallel programs written for the MPI and PVM libraries. Condor is suited to both heavyweight and lightweight grids.

These are not the only two types of grids in existence, but rather a simplification of the two most common categories that are used for running traditional grid applications. Another type of grid is the emerging *services model*, in which specific functionality is provided through well-defined interfaces and accessed through standard protocols. Services can be provided by any type of machine, ranging from supercomputers and high-end servers down to individual workstations and handhelds. Because access occurs through standard protocols, clients require no knowledge of the platform that the server is running. Service-based grids such as these are common in enterprise environments, where they are used to integrate a variety of databases, legacy systems, and other business applications in a standard manner. A good example of this type of usage is Oracle's BPEL Process Manager [85], which allows users to graphically construct a work flow used to integrate services in a coordinated manner. Similar projects, such as Taverna [83], are targeted towards scientific disciplines such as bio-informatics. CORBA [82] is particularly well-known system based on this architecture, which models services as objects that can be accessed remotely through method calls.

The recent popularity of peer-to-peer architectures [86] provides many useful insights into the ways in which large-scale distributed systems can be effectively built. Traditional approaches to grid computing, particularly those concerned with heavyweight grids, have relied on systems that require extensive installation and configuration efforts, and often rely on manual specification of many system parameters. Often these are registered with a central authority or hierarchical system such as a directory service, an action which requires appropriate authorisation from other parties. Many peer-to-peer systems, on the other hand, have successfully been deployed across the Internet on the scale of millions of nodes, with little or no system administration effort required. All that is involved in connecting a machine to such a network is for a user to install a simple piece of software, which then connects to a few other nodes running on other people's desktops, and from there is able to access either the entire network or large parts of it. This approach to distributed systems design has received little attention in mainstream grid computing research, which has largely concentrated on the former approach. We argue that such architectures are applicable to grid computing. While a full investigation of peer-to-peer grid computing systems is outside the scope of this project, we intend to consider this approach as part of our investigation of scheduling strategies, as distributed mechanisms of this type are particularly suited to the scalability requirements of grids.

2.2 Execution Models

A grid computing system supports one or more *execution models*, which dictate ways in which computation is performed, data is accessed, and the operation of applications and resources is controlled by users and system administrators. While the grid itself provides the underlying infrastructure, an execution model describes the way in which this infrastructure is used. Some grid systems are designed with one particular execution model in mind, while others allow a range of different models to be used. The flexibility to choose an execution model appropriate to one's needs is a desirable property of a grid, and numerous existing grid systems allow this.

Grid execution models have evolved from previous computing paradigms such as batch processing, client-server, mainframes, and the world wide web. They have been adapted for use in distributed environments, and have proven to be very useful approaches for running applications in such environments. An overview is provided here of the two most common models, *job submission* and *services*, and the differences between them, as well as ways in which they can be combined.

2.2.1 Job Submission

In the job submission model, a client submits an application to one or more resources for execution. The application may be run immediately or at some future point in time. The client is notified upon completion, and is able to retrieve the results of execution such as output and any data files the application has created or modified. When submitting the job, the client supplies the executable file, or the name of a program already installed on the host, along with any other input files and information about how to run the application, such as environment variables and command-line arguments.

The main use of the job submission model is to allow users to run applications on more powerful systems on the grid than they have access to locally. Instead of executing programs on their local workstation, a user may send job submission requests to remote machines on the grid, which enable the application to complete sooner, or multiple instances of the application to be run concurrently on different machines. This approach has similarities with traditional batch processing systems, in which a large, powerful computer is responsible for running jobs provided by users, except that instead of just the one machine, a user has access to potentially large numbers of hosts distributed across the network.

When choosing a machine to run a particular application, the user must submit it to a resource that provides the necessary capabilities for executing the program. These requirements generally include the hardware platform and operating system, as well as other software dependencies such as shared libraries and external programs that are used by the application. In some cases, dependency on specific hardware or operating systems can be avoided through the use of an intermediate layer such as a virtual machine, however this still constitutes a platform on which the application runs, and must be provided by a machine in order to run the application. A job can only be run if all of its requirements are met; this restricts the set of machines on the grid that can be used for a particular job to the subset that satisfy the requirements. Even if the job submission request references an already-installed executable file instead of uploading one, it still requires platform and installation dependent information such as the path name of executables, configuration settings, and environment variables.

There has been a large number of different job submission systems developed for use on clusters, many of which are reviewed in [11]. Some of the main features supported by many of these are as follows:

- *Load balancing* [67] involves allocating jobs to cluster nodes such that the amount of machine load is evened out between them. Assigning all jobs to the one machine is likely to heavily overload it; spreading them out appropriately means that there is an approximately equal amount of processing work assigned to each machine. A cluster controller must monitor the load of each machine, and offload work to other nodes if one is becoming overloaded. This improves the performance of the system and reduces application run times.

- *Checkpointing* [72] allows the complete state of a process to be saved for disk so that it can be continued later from the point at which the checkpoint is taken. This allows a process to be stopped, and then resumed at a later point in time by recreating the process based on the saved state. This can happen as a result of a request to pause a processes, or if a system crash occurs during a long-running computation, avoiding the loss of all work done during that computation.

- *Process migration* [41] is where a process running on one machine moves across to another machine and continues execution. It is particularly useful for load balancing; if a machine becomes too heavily loaded, then some

of the processes on it can be migrated to other machines. It can also be used in situations where a particular machine needs to be shut down for maintenance purposes; any programs that were running can continue elsewhere after the machine is taken offline. Process migration is sometimes implemented using checkpointing; the state of a process is saved to a file, copied to another machine, and then used to restart the process on the new machine.

- *Job suspension* [65] involves checkpointing the state of a process to disk, and storing it somewhere so that the job can be resumed at a later point in time. It is sometimes used when jobs have been started on an user's workstation while it was unattended, and the user has returned and wishes to use it once again. It means that the computation being performed by the machine can be paused, and then continued at a later point in time, such as when no user activity has been detected for some period.

- *Runtime limits* can sometimes be specified for jobs to ensure they do not consume more than a certain amount of CPU time. This can be used my administrators to control the way that resources are used and to prevent runaway applications from taking up too much of the system. Usually, an application will be either terminated or suspended after it has reached its allocated CPU time. Limits on other types of resource usage such as storage and network traffic may also be supported.

Systems which allow for jobs to be submitted and assigned to resources are called *resource managers*. They take care of all the low-level details such as receiving and parsing job submission requests from users, copying files between client machines and individual nodes on a cluster or grid, performing the appropriate authentication and authorisation required to run a job, monitoring the execution of jobs and providing feedback to administrators, and many other aspects. The specific functionality provided by the resource manager varies a lot between different packages; sometimes this functionality is provided by external pieces of software, and sometimes other functionality normally provided by separate systems is included in the resource manager.

PBS (Portable Batch System) [56], originally developed at NASA, is designed for providing flexible mechanisms for executing jobs on clusters. It provides extensive control over scheduling policies by allowing administrators to plug in different scheduling algorithms, and these can be implemented in any one of several different languages. PBS also provides support for routing jobs from one location to another, and performing staging of data files to remote hosts before executing jobs that depend on those files. TORQUE (Tera-scale Open-source Resource and QUEue manager) [31], based on PBS, adds numerous additional features such as better fault tolerance, extra scalability, better logging facilities, many bug fixes, and provides better integration facilities for schedulers.

LSF (Load Sharing Facility) [89] provides a transparent view of a collection of different machines. It allows jobs to be submitted and run in a variety of different modes, including sequential and parallel jobs, as well as interactive or batch modes. It achieves transparency by running applications in an environment that appears almost identical to that from which they were submitted, avoiding problems caused by different file locations, user names, and other environmental attributes. Fault tolerance is also provided for all jobs that are submitted.

IBM's Load Leveler [61] supports both clusters and multiprocessor machines, and provides a similar feature set to that of PBS and LSF. It contains a central scheduler to which all jobs are submitted, and then distributed to nodes in the cluster. New nodes can be added to the cluster dynamically, and the system will be able to make use of them. Users of individual workstations can specify when and how their machine is available for use by the cluster management system. Load Leveler also supports the job submission syntax of the older NQS system [93].

Condor, mentioned previously, is designed as a system for managing execution of jobs on a cluster or network of workstations. It uses a job submission system which allows users to supply many job parameters and an executable file, which is sent to an appropriate machine for execution. It matches jobs to machines based on the requirements specified by the job and certain conditions specified by each resource. It also includes a number of different types of execution environments, which support different features such as checkpointing, process migration, and parallel programs.

A widely used resource manager for grid environments is GRAM (Grid Resource Allocation Manager) [36], part of the Globus toolkit. GRAM provides an interface to which users can submit job requests specified in RSL (Resource Specification Language). The syntax of RSL allows information such as environment variables, command-line parameters, input files, and other details relevant to the job to be specified. The name of the executable can either be given as a file that already resides on the server, or a file on the client machine that is transferred via the network before execution. A successor to GRAM, included with the Globus toolkit, is MMJFS (Master Managed Job Factory Service) [104], which provides similar functionality using a web services interface. Both of these are intended to act

as front-ends to local job submission systems such as PBS, LSF and others described above, for clusters and parallel computers that are made available as nodes on the grid.

While GRAM and MMJFS allow jobs to be submitted to individual grid resources, by themselves they do not support submission to multiple resources. Co-allocation [37] is a technique often used when multiple resources are required in order to run a job. It involves negotiating with the relevant resource management systems to get guarantees about what resources will be available and when, and ensuring that these are in place before an application is started. Advance reservation [46] mechanisms can be used to obtain these guarantees, and are particularly useful in situations where starting a job must be an atomic operation; it is often undesirable to have some parts of a job running when others have failed to start up. The Globus toolkit includes a co-allocator called DUROC (Dynamically Updated Resource Online Co-allocator), and is particularly useful for supporting parallel programs within a Globus grid.

The responsibility for deciding which resources will be used to execute jobs and when is generally handled by a *scheduler*, which is a separate entity from the resource manager. Some of the resource managers listed above provide extensive support for plugging in different scheduling systems. The resource manager is essentially a low-level mechanism for using the grid; a scheduler is a higher-level entity which provides more overall functionality, such as providing a more transparent view of a collection of resources to a user, by hiding a lot of the specific details of the resources. Grid schedulers are discussed further in Section 2.5.2.

2.2.2 Services

The services model is similar to that of client-server computing, where a server provides the ability to perform certain processing on behalf of a client, and the two communicate together to invoke operations. A server provides one or more services, each of which contains an interface with operations that can be invoked by a client using a messaging protocol such as Sun RPC [106], XML-RPC [125], DCOM [102], MSMQ [39], or CORBA [82]. At the programming language level, these operations are typically invoked using method or function calls in a similar manner to the way in which local procedures are used. These are translated by the runtime system into messages that are sent over the network to the server, which executes the relevant code and then returns a response to the client. Operations can either by invoked *synchronously*, where the client pauses execution while waiting for the operation to complete, or *asynchronously*, where the client continues executing while the operation is performed in the background.

Because the client and server interact only using standard protocols, the client needs no knowledge of the server's platform or implementation details of the service. Additionally, the server does not need any information about how the client operates, other than the knowledge that it speaks the protocol. Therefore, the client and server can interact with each other in a platform-independent manner. A client can access different implementations of a service on different machines in exactly the same manner, as long as they conform to the same interface. The operations included in an interface, and the data types that they expect, are declared in a description language such as IDL [81].

In general, the set of services provided by a particular machine is decided by the administrator of the machine, who must specifically install and configure the desired services [69]. All of the executable code necessary to implement the service resides on the machine for the entire lifetime of the service instance, and is executed in response to requests from clients. A separate software component provides the necessary interface to the service implementation, translating operation requests received via the network into the actions and data necessary to perform an operation on the machine.

Services often reside within a middleware frameworks known as *service containers*. These provide all the necessary support mechanisms such as protocol support, security and manage-ability. Administrative functions include deploying, configuring, starting, and stopping services. Examples of service containers include Apache Axis [8], Oracle Application Server [84] and IBM's WebSphere [14]. While most of these systems rely on administrator-controlled setup and installation of services, the notion of reconfigurable, component-based middleware has been proposed as an architecture which extends traditional middleware to be more dynamic [33]. Systems build around this model would more easily support the addition of new services and configuration changes to them, without necessarily requiring administrator intervention. Dynamic service deployment is discussed further in Section 2.2.3.

The most prominent model of services at present is *web services* [24]. This model is based on the world wide web, where clients and servers can easily communicate with each other in a platform independent manner. However, unlike the traditional view of the web where largely unstructured content is accessed by human users, web services involve client programs accessing remote services that implement well-defined functionality. Service interfaces specifying sets of operations that can be invoked by clients are defined using WSDL (Web Services Description Language) [126], and messages are exchanged using SOAP (Simple Object Access Protocol) [127].

The web services model has been extended to Grid Services [47], which provide additional functionality desirable in a grid environment such as security, authentication, service instantiation, and lifetime management. The OGSA model [47] defines an architecture in which clients can *instantiate* grid services, which remain in existence over some period of time, and eventually get destroyed either at the request of a client or due to a timeout. Clients interact with a service instance through a GSR (Grid Service Reference), which contains all of the information required to send messages to the service. At the programming language level, these are mapped into specific constructs such as objects or pointers. OGSA also specifies mechanisms by which clients can subscribe to services and receive notification of certain events that occur. The OGSI (Open Grid Services Infrastructure) specification [116] defines implementation details of the architecture in terms of specific service interfaces and operations. Current implementations of the standard include Globus 3, OGSI.NET [123], and pyGridWare [40].

The recently-proposed WSRF (Web Services Resource Framework) [42] re-factors many of these concepts into *WS-Resources*, which are similar in nature and functionality to grid services [34]. The main difference is that where a grid service represents a service and its state, a WS-Resource considers these to be separate entities, but represents a pairing between them. A given web service, provided by a machine, contains internal functionality and a set of operations that can be invoked by clients, but it does not maintain any state. Instead, this is captured in a stateful resource, an implementation-defined entity that is used by the web service whenever an operation is invoked. A WS-Resource thus represents a stateful entity and the operations which can be performed on it. This specification is being implemented in the upcoming Globus 4, WSRF.NET [60], and also pyGridWare.

A number of other service-based grid systems have been developed prior to the standardisation efforts around web services. NetSolve [21] uses services implemented in native libraries and makes them accessible via a custom protocol. Clients wishing to access these services must link with the NetSolve client libraries which provide a programming language interface for invoking remote operations, and support for the protocol used to communicate with the services. Requests are sent from clients to a central authority which uses a load-balancing technique to select a specific resource from the list of those which provide the service. The request is then routed to that resource, where the relevant function in the shared library is invoked to perform the requested operation. The interface used by clients to access services is simple, but provides only limited functionality - data types are restricted to scalars, vectors and matrices. While only NetSolve libraries can be used to access the services, the protocol is platform independent, allowing clients to access different implementations of a service. Ninf [97] is a similar system which provides more or less the same functionality; however, it allows services to be implemented as separate executable files, making deployment easier.

2.2.3 Hybrid Approaches

The key difference between the job submission and services models is that in a job submission system, the client generally needs platform-dependent and installation-specific knowledge in order to use a remote machine. This most often involves the specific execution environment provided by the server, if the client is sending an executable to be run, but can also include details such as full path names of files on the server, specific command line arguments that must be supplied, and values of environment variables that need to be set. This difference has significant implications for the range of functionality that can be provided by a given host, and the ease with which heterogeneity can be supported. Most existing uses of grids focus only on one approach or the other - either treating the grid as a pool of arbitrary resources that can be used to run any type of application, or a collection of services that reside on particular machines and are used for specific purposes. There are, however, ways in which the two approaches can be combined.

Job submission services provide a service which accepts either program executables or pathnames of executables, as well as other details relevant to running the program, and then launches execution of the program. This uses a service interface accessed through an RPC protocol to transfer the program and data files and other information from the client, and job submission and status querying are treated as operations that can be performed on the service. This represents a case of implementing job submission *on top of* the services model. An example of this is MMJFS [104], a grid service included with Globus 3 which implements a job submission mechanism.

Dynamic service deployment is where the code for a service is supplied by a client, instead of already residing on the remote machine, but then the client interacts with the executing code through a service interface. Once the service is launched is accessible in the same manner, and the client makes RPC calls to perform specific operations which in turn translate to execution of certain portions of the submitted code. This represents a case of implementing services *on top of* job submission. An example of a system supporting dynamic service deployment is OGSI.NET, which contains a *meta-factory service* which accepts as its parameters one or more .NET assemblies that implement a service, and then instantiates that service by executing the submitted code.

Hybrid programs are those that use both job submission and services to perform their processing. An application resides on the client machine, from which it submits individual jobs to other machines to perform certain portions of its work, and invokes operations on services to perform others. The programming model used in this case would provide mechanisms for both; for example, creating a new thread and having it placed on another host, and allowing remote services to be accessed through method calls. This represents a case of implementing job submission *along side of* services. An example of a hybrid program would be a Java program that uses the CoG Kit [119] to access resources via the Globus infrastructure to submit jobs to remote hosts that perform part of its computation, and make calls to grid services to perform other operations. Hybrid programs such as this are of most relevance to the research proposed here.

Finally, programs which are submitted to a job submission may also access services while they are running. In this case, the "client" is actually the host that is executing the submitted program, while the "server" is the machine that provides the service that the program connects to.

2.3 Parallel Processing

In the MIMD (Multiple Instruction Multiple Data) architecture, a *parallel process* consists of multiple, independent execution streams of instructions, called *tasks*. The tasks can be executed by different processors, possibly residing on different hosts. The tasks interact with each other during execution. When they are distributed across machines, this interaction is in the form of messages sent across a network. The *process* here is a logical grouping of tasks that are interacting with each other. Some part of the system has knowledge of all of the tasks associated with a process and where they are being executed, and coordinates operations such as creating new tasks and assigning them to hosts. From the point of view of a parallel processing system, a *sequential process* is simply a special case in which there is only one task.

In a distributed system capable of executing parallel programs, it is generally possible for each host to concurrently execute multiple tasks from the same or different processes. Figure 1 shows an example of a system with two processes, each of which has a set of three tasks distributed across three hosts. Each host executes a task from each process. In this situation, the part of the system that maintains task and process information must also support the concept of multiple concurrent processes, and distinguish them from one another. Each task belongs to a particular process, and its lifetime is restricted to be within that of the process. Tasks only interact with other tasks in the same process.

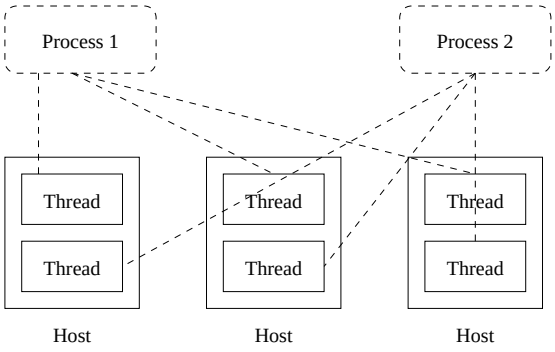


Figure 1: Multiple parallel programs running on a set of hosts

When a host has one or more tasks running on it, it is necessary for it to provide some mechanism for these tasks to interact with entities external to the hosts, such as other tasks, and the part of the system managing process execution. This interaction may be for purposes of inter-task communication, data access, or administrative functions such as launching or terminating other tasks. The host must must provide an external interface which provides a method of communicating between these external entities and the tasks. The interface may provide other functionality such as identifying which tasks reside on a machine, or creating new tasks on the host.

A central component of the system maintains information about all processes that are running, which tasks are associated with them, and where those tasks reside. It is responsible for interacting with the hosts to send data to or from tasks, and to create, terminate and possibly migrate running tasks from one host to another. Tasks running on one host may use this component to communicate with other tasks, or connect to the external interfaces on the other

hosts directly. The specific functionality of the component, and the semantics by which tasks are communicated with and controlled on the hosts varies between different types of parallel, distributed systems.

In job submission systems used for parallel computers, each job normally corresponds to a process consisting of multiple tasks. The set of tasks is monitored, and once all of them have completed, the job is considered finished. In this way, the user can manage the running of a process as a single entity, despite the fact that internally it consists of multiple concurrent components. Another view, more common in cases where there is no interaction between tasks, is to consider each task as a separate job. In this situation, the parallel "program" is normally a script or set of commands to the job submission to launch the same or different programs with different parameters, and the output is stored in separate files which are collated or analysed later. This approach is also used sometimes when running parallel programs on a grid system that does not directly support parallel jobs.

The concept of processes and tasks can also be used within the context of the services model. In this case, there is a parallel or sequential process consisting of one or more tasks, and also services that are being accessed. The services are essentially like tasks in the process; the main difference being that the code to implement them already resides on a particular host and they are accessed via a remote procedure call-style messaging protocol. Despite the different instantiation and communication mechanisms used for services, the computation performed by them is still essentially part of the parallel process, because the code is being executed on behalf of the process.

2.4 Data access

All grid applications rely on data in one form another, whether it resides in data files, databases, or other forms. Often, the amount of data needed by an application is large, and requires significant communication between the program and the file storage location. In some cases the data may be accessed locally, and in other cases it may be accessed remotely. Many different types of storage mechanisms and network file systems exist for accessing data.

When running an application on the grid, it is desirable to keep the computation close to the data, to minimise the amount of time that the application must spend while waiting to read and write from files. This is particularly true in a wide area network environment where there may be slow data links, resulting in high overheads for accessing large files located on another machine. Just as when using a job submission or service model to perform processing it is necessary to select an appropriate machine to use, it is also necessary to select a suitable place to store the data that makes it easily accessible to applications.

The most common method used by applications to access data files is through native operating systems APIs or equivalent constructs provided by the programming language, such as C's `fopen()` function or Java's `File` class. These can be used to directly access any file residing on the filesystem of the machine in which the program is running, such as a local hard disk or removable media. Many operating systems allow remote file systems to be *mounted*, which imports the namespace of an external storage system into the local one under a particular prefix, such as a mount point under UNIX or a drive letter under Windows. Common protocols for accessing remote file systems in this manner include NFS [96], NCP [77], CIFS [70], and AFP [9]. Many distributed file systems, such as AFS [98], Sprite [79], xFS [7], and Coda [99] can also be accessed in a similar manner.

Not all remote data access protocols are commonly supported by operating systems as mountable file systems. If files are residing on a remote server, and it is not possible to access them through the operating system's normal file access mechanisms, it is necessary to connect to the remote server either directly, or indirectly through some other layer of abstraction. A common way of doing this is to use a library linked with an application that implements the client-side of the protocol, and provides functions accessible to a programmer which allow files to be accessed. Examples of these include Java's `URLConnection` class [108] and Condor's Chirp library [32]. In some cases these libraries support a range of different file access protocols, such as the GASS client library [17].

Another way for an application to access a remote data source is through the use of an *interposition agent* [63], which provides a layer between the application and the operating system, and intercepts API calls, providing an alternative implementation which redirects the operation to a remote machine. For data access, calls such as those used to open and read from files are replaced with implementations that connect to a remote server and transfer data, instead of opening a file on the local file system. Interposition agents can be implemented at user-level, and provide an easy, transparent mechanism for redirecting normal file access calls to a remote machine, making them an attractive way of enabling legacy or commercial applications to make use of grid technologies without recompilation or modification [113]. An example of a remote file access system implemented in this manner is Parrot [114].

The term *data grid* [27] is used to describe grids that are designed with the goal of supporting applications which require access to very large amounts of data. Often these involve read-only access to data sets produced from scientific

experiments. The approaches to data storage in such a grid are different to the traditional approach of simply accessing a file system on a single remote machine - files in a data grid are distributed across multiple machines in different parts of the network, and often multiple copies, or *replicas*, are present. This enables a much larger amount of data to be stored than would be possible on any single machine, and replication allows for fault tolerance and faster access, by allowing a user to access the replica that is closest to them on the grid.

An example use case of a data grid is the upcoming high energy physics experiments that will be performed using the large hadron collider currently being constructed at CERN, due to go into operation in 2007 [59]. The collider will produce several petabytes of data per year, which must be made easily accessible to thousands of physicists all around the world. It would be impractical for all of these users to access one copy of the data residing in a central location; the demands on the network and hardware containing the data set would be enormous, and access from distant locations would be slow. Instead, separate copies of the data set will be stored at key locations in different geographical areas around the world, and relevant subsets of it will reside at individual institutions. A physicist wishing to access some portion of the data will do so using one of these replicas, either by copying the relevant data to a local machine, or by arranging for the processing to be performed on another machine which has high-speed access to the data.

2.5 Scheduling

Scheduling, in its most general form, refers to the allocation of resources over a period of time for specific purposes. Decisions about which resources to use and when are made based on information about the resources themselves, and the ways in which applications wish to utilise them. The objective when performing scheduling is to optimise for one or more variables, such as job completion time or resource utilisation. The term *schedule* refers to a set of resource assignments, either at a fixed point in time or over a period of time, and is the result of the processing performed by a *scheduler*. The nature of these assignments, and the way in which they are derived, varies greatly between different types of systems. Three types of scheduling are discussed here: *computation scheduling*, *data scheduling* and *service scheduling*. Computation scheduling is considered both in the case of traditional parallel computing systems, as well as emerging grid technologies.

2.5.1 Computation Scheduling on Parallel Computers

Much of the previous work in the field is targeted at parallel computers that consist of either tightly-coupled shared memory machines with many processors, or clusters consisting of a large number of commodity, off-the-shelf machines. These architectures differ from grids significantly in that they have high-bandwidth, low latency communications links between them, and are all under the central control of a single authority. Scheduling in this context is a simpler problem because the scale of the machine is limited, and the scheduler has complete control over every program that is executed. This is significantly different from a grid environment, where there may be additional work being performed by resources at the discretion of individual owners.

A good overview of scheduling algorithms for tightly-coupled parallel computers is given in [26]. All of these algorithms give jobs *exclusive* access to processors - each processor can be executing at most one job at a time. Usually, a job must run to completion before the processor(s) it uses become available again for other jobs. Sometimes, preemption is supported, in which a job is suspended temporarily to allow another job to use the processor for a short period of time. The exclusive model has been used in a large amount of parallel computing research, and is promoted as being a good choice due to the fact that it is simpler to consider one job per processor than multiple jobs per processor, and because keeping only one application in memory reduces the extent to which virtual memory is required, avoiding the associated performance penalties.

An alternative model is where jobs have *shared* access to processors, and the multi-tasking features of the underlying operating system provide the ability to run multiple multiple jobs on a processor using time slicing. This is less common in clusters and shared-memory parallel computers, but can be useful for a number of reasons. One is that higher processor utilisation can be achieved by allowing tasks to be executing while others are blocked on I/O requests or synchronisation points [44]. Another reason is that submitted jobs do not need to wait for processors to become free before starting - this is particularly beneficial when attempts are made to run a short job when all the processors are busy executing a long-running job. Our proposed research takes into account both exclusive and shared processor access.

The choice between exclusive and shared processor usage has important implications for the scheduling algorithm. With the exclusive approach, jobs can only be run when there are enough processors available to execute them, meaning that the start time of a job may be delayed significantly. In this model, a schedule can be represented graphically as

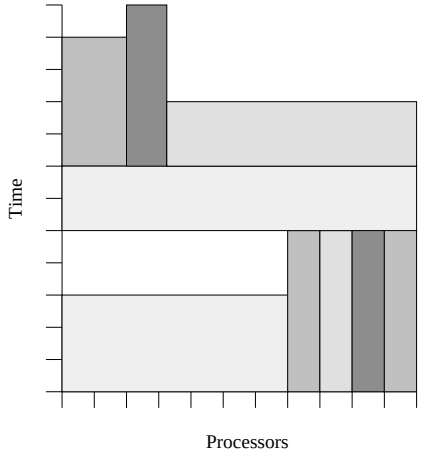


Figure 2: Graph of CPU usage in a parallel computer

a 2-dimensional graph, with the columns representing processors and the rows representing periods of time, as shown in Figure 2. Each cell is shaded according to the job that is assigned to the relevant processor during the corresponding time period. This visual representation allows for the processor utilisation to be easily seen by looking at how much of the grid is shaded. Often, there will be "holes" in the schedule corresponding to processors that go unused for some time because no job can be assigned to them. These occur when there are jobs in the queue which require more processors than are available during that time period. A technique called *backfilling* [130] is often used to allow short jobs to move ahead in the queue to utilise these processors, as long as they do not cause the delay of other jobs that reside before them in the queue. In grid environments containing these types of parallel computers, process migration can sometimes be used to offload work to another machine that does have enough processors available to provide exclusive access.

In the shared model, the assignment of jobs to processors can be represented as a 3-dimensional graph. Columns and rows still represent processors and time periods, but the third axis shows multiple jobs being assigned to a processor during a given time period. Because most multi-tasking operating systems allow either a large or theoretically unlimited number of concurrent tasks to execute on a processor, the third axis can extend as far back as possible to represent as many tasks as necessary. In such a schedule, the "holes" sometimes found in 2-dimensional schedules are not present, as jobs can be started immediately, instead of being delayed and processors therefore going underutilised.

Many scheduling algorithms devised for parallel computers rely on knowledge about how long an application will take to execute. These are generally supplied by a user, who must estimate how long their job will take to run, by methods such as doing a test run with a smaller data set and multiplying the time appropriately. These time estimates are only reliable given sufficient knowledge of the hardware, and the way in which the application performs on it. Obtaining these estimates can be a non-trivial task, and sometimes users may supply less accurate estimates due to the difficulty of determining them properly [78]. While it is often possible to predict execution times for programs on parallel computers where the scheduler and job submission system is tightly integrated with the computer, it is far less practical on a grid, where resource capabilities differ significantly, and many other factors such as network utilisation, machine failures, and other workloads can influence the performance of an application. Therefore, the research proposed here does not rely on this information.

A major reason why scheduling on a parallel computer is simpler than that of a grid is that a parallel computer has a central point of control. The scheduling and job submission mechanisms have complete knowledge of and authority over all of the individual processors, and can make assumptions about their availability and capability to execute jobs. All work that the processors do is under the control of this single scheduler, which does not have to contend with other activity outside of its control. Additionally, the scheduling algorithm only needs to scale to the number of processors in the parallel computer, which is limited to a few hundred or possibly a few thousand, even in the largest computers.

In contrast, a grid potentially has a much larger number of processors, none of which are under the complete control of the grid scheduler. Each resource in the grid may also be used by its owner for other purposes, and be performing activities that are not visible to the rest of the grid. The fact that millions of processors may potentially be available also requires the use of more scalable algorithms, and the need for high availability necessitates distributed implemen-

tations. Algorithms designed for parallel computers cannot be directly applied to a grid environment, because many of the assumptions they rely become invalid.

2.5.2 Computation Scheduling on Grids

In a parallel computer, the location and start time of jobs is determined as part of the same process, as the entire computer is under the control of a central authority. In a grid, however, the process is more complicated because each individual node has autonomy. In this situation, the problem is split into two parts: *local scheduling* and *metascheduling*.

A local scheduler resides on each computational resource on the grid, and, when it receives a job to be executed on that resource, decides when the job will actually be executed. Some local schedulers only permit one job at a time to be executing on a particular resource; others that are submitted while it is busy will be put in a queue and started once the job completes. Other local schedulers allow more than one job to be running on a resource at a time, by using the multi-tasking features of the underlying operating system. In the latter case, a job submitted to a resource will start immediately and run concurrently with the other jobs.

A metascheduler, or *resource broker*, makes decisions about which resources will be used to execute a particular job. When a job is submitted to the resource broker, it uses information about the available resources and possibly other running jobs to decide when and where the job should be executed. It may postpone execution until one or more resources becomes available, or start the job immediately on a resource that is chosen according to some criteria, such as lowest amount of load. A metascheduler may assign the different tasks of the job to different resources, and possibly migrate tasks from one resource to another while running, if this is supported by the execution environment. The problem of determining optimal task placement for a set of hosts has shown to be NP-complete, so heuristic-based approaches are generally used [18]. In cases where a parallel computer or cluster is represented as a single resource on the grid, a local scheduler may also be responsible for assigning tasks to specific processors, in a similar manner to the way in which the metascheduler assigns tasks to grid resources; this is effectively a second level of metascheduling.

There are three phases in the process of metascheduling, as described in [101]. The first phase, *resource discovery*, involves creating a list of candidate resources from all of those available on the grid, based on the ability of those resources to execute the job. All of the resources in this list must meet certain minimum requirements, including matching the correct platform, and providing the user with the authorisation to use them. The second phase, *system selection*, gathers information about these resources and figures out which ones are likely to provide the optimal execution time for the application. They are usually ranked in order of execution time, or some other related parameter, and the top ranking one is chosen. In the third phase, *running the job*, the program executable and other related information is actually sent to the resource, which then executes it. In some systems, launching of a job is considered separate from the process or scheduling itself, and in others the job submission mechanisms are tightly integrated with the scheduler. *Rescheduling* is the process of repeating these steps and then migrating the task to a new location. This is particularly useful if the load on the machine running the task becomes heavy, and performance gains could be obtained by using a different host.

Metascheduling may be done at the application level, or globally. *Application-level metaschedulers*, such as used in Nimrod/G [1], AppLeS [15], and MARS [50], are run whenever a job is to be submitted by a user, and operate by selecting resources based on information available about those resources, such as their network connection speed, CPU load and other parameters. They then make the choice about where to run the application based on which resource appears to be the optimal choice. Details of other applications that are already running, or being concurrently submitted, are ignored. *Global metaschedulers*, such as those included in GrADS [117] and Utopia [129], on the other hand, take into account information about resources *and* other jobs. Because they posses a wider range of knowledge about what is happening on the grid, they can make more informed decisions, thus resulting in better application performance and resource utilisation. This can be important in situations where information about other applications is relevant to making the scheduling decision. For example, with application-level schedulers, a large number of jobs submitted at the same time could all be assigned to the one host, if this host appears to be the best choice just by looking at its properties and not the other jobs. But a global scheduler would recognise that the jobs need to be more widely distributed in order to obtain good performance. For this reason, the research proposed here will focus on global metascheduling.

Metaschedulers can operate in either a *centralised* or *distributed* manner. A centralised metascheduler [128, 48] consists of a central point to which all jobs are submitted, and uses information about all hosts available on the grid to make decisions about which hosts should be used for which jobs. A distributed metascheduler [25, 54, 16] divides the responsibility for job scheduling among multiple hosts on the grid, providing fault tolerance and scalability. However,

distributed algorithms are more complex than centralised algorithms, and so implementing a distributed metascheduler is a more challenging task. Centralised schedulers are still relevant in many situations where the size of a grid is limited, such as within a single organisation or set of collaborating institutions. The research proposed here will look at both of these types.

Most existing metaschedulers are designed for sequential jobs. When an application consisting of multiple tasks is run on the grid, each task is often scheduled as an independent job, without taking into account its relationship to other tasks. This can be problematic if two tasks that interact with each other during execution are placed on separate machines, and the network connection between them is slow. Local schedulers which put the jobs on separate queues and start one before the other can also cause problems, because the first one may need to wait until the other starts before performing the main part of its computation. For this reason, knowledge about the structure of a parallel program needs to be used by the metascheduler when making placement decisions. Interacting tasks should be placed close together on the grid, and started at the same time; this is only possible with appropriate direction from the metascheduler. These aspects of scheduler design form part of our proposed research, further detailed in Section 3.

A trade-off exists between the degree to which a grid middleware system allows resources to be independent and the ease of making good scheduling decisions. If the scheduler was to have complete knowledge of and control over every aspect of the individual machines, it could guarantee that the performance of the programs it schedules will not be affected by outside influences that it cannot predict. However, this level of control is impractical on grids where owners of individual machines still want to have the ability to run other applications and control the way in which their machines are utilised by the grid. But if the scheduler has too little information to work with, for example if it does not know what other jobs are running on the machine, then the resulting scheduling decisions are of much less quality [49]. When designing middleware, it is necessary to pick a point along the spectrum that provides enough information and functionality for the scheduler to operate effectively, while still allowing each machine to maintain a certain amount of control over what can run on it and when.

One way in which the autonomy of individual resources has consequences for the scheduling mechanisms is how clusters are integrated with a grid. A cluster may be represented as a single resource, with all jobs sent to the local scheduler on the front-end machine, which then assigns it to one or more nodes in the cluster. Alternatively, each individual cluster node may be individually accessible on the grid, so that the scheduler sees a collection of resources comprising the cluster, and decides which nodes to use itself. The latter approach gives more power to the metascheduler, because it has more information to deal with. If the cluster is represented as a single resource, this complicates the scheduling of parallel programs, since the metascheduler must have specific knowledge that multiple machines reside in a single resource in order to send multiple tasks to it. The front-end must also ensure that if the machines can only be used by one task at a time, then it communicates only the number of *available* nodes to the metascheduler at any given point in time, and that the failure that occurs in a single cluster node is not interpreted as failure of the entire resource.

Metascheduling can also make use of knowledge about the structure of an application. Master-Worker style applications are relatively simple to schedule, with the clear division between work units and explicit communication patterns [57]. Directed acyclic graphs specifying the dependencies between components of an application make the ordering explicit, so the scheduler knows when that particular tasks can begin execution only after certain other tasks have completed [3].

Another type of information that can be useful to a metascheduler is the details of the network topology and performance [107]. By considering the amount of bandwidth available between different resources in the grid, and the communication requirements of an application, choices can be made in such a manner that the need for computational power is balanced with the need for fast data transmission. Jobs which are compute intensive can be scheduled to faster processors even if the network connections to them are slow, while parallel programs which contain a lot of synchronisation or data transfer between tasks are generally better off on machines connected at higher speeds, even if they have slower processors. Locality of tasks also has an impact on performance [111]; tasks placed on machines connected via a local area network are able to communicate with each other more efficiently than on machines distributed across a wide geographical area. Reducing the level of parallelism so that it can execute on a smaller number of well-connected machines is sometimes a worthwhile trade-off to make.

2.5.3 Data Scheduling

Data scheduling is the process of making decision about where to place data files that are accessed by applications. In order to obtain good performance, it is desirable to have the data files accessed by an application residing close to where the application is run, preferably on the same machine or another to which it has a high-speed network

connection. For this reason, scheduling applications to run on hosts purely based on an estimate of the computational cost is less effective than also taking into account the costs associated with data access [91]. By using scheduling algorithms that attempt to achieve this locality, data can be accessed faster and the load on network connections is reduced.

The simplest form of data scheduling involves copying all of the required files to the host that has been chosen to run a job. As part of a job submission request, the user must specify all of the files that are needed by the application. The files are then transferred to the remote host before the job starts running, and the program can then access them through the local file system. Once the job has finished, any files that were created or modified are copied back to the original location. This approach is used by the job submission system in the Condor project [115], which allows file sets to be specified as part of the job submission request. Copying data files whenever jobs are run effectively ties the data scheduling to the computation scheduling, rather than making it a separate process. Other mechanisms can provide more efficient placement strategies which optimise data across multiple jobs.

An example of these strategies useful for multiple jobs is that of pre-staging, where data files are placed on a host before running a series of jobs which all rely those files for input. This can result in significant performance gains compared to transferring the files each time, if there is a large amount of data or many jobs are being run. This has been shown to be useful in particular for parameter-sweep operations, which involve a large number of jobs being run with differing parameters, but which often use the same input files [22].

Data on the grid can be replicated across different sites [118]. Having multiple copies of a particular data set allows a larger number of sites to have high-speed access to the data, increasing the range of possible compute resources that can be chosen to efficiently execute a job that depends on that data, thus improving load balancing [92]. Creation of replicas at various points throughout the grid is particularly useful for read-only access [87]; however, this mode of access cannot always be assumed [88]. Cases involving read-write access to the data require mechanisms in place to ensure the consistency of the different copies, such as the techniques described in [53]. Replica management constitutes part of the data scheduling problem, because decisions must be made about when and where to create replicas. By placing replicas in suitable locations, programs can have efficient access to data, reducing their execution times.

One way of creating replicas is to do so manually, based on developer or user knowledge of the ways in which applications access data. However, this can be cumbersome and time consuming, and so automated replication mechanisms have been proposed as a desirable alternative. Directory-based replica management schemes have been proposed in which all files and replicas distributed around a grid are registered with a central *catalog* when they are created, and this catalog is searched by clients when attempting to access a particular file in order to find the most suitable replica [4].

Peer-to-peer file sharing systems such as FreeNet [30], Gnutella [2], PAST [95], Kazaa [52], and others have demonstrated scalable, efficient means of creating file replicas without any central point of control. These strategies have been applied extensively for the purpose of trading music, software and video files over the Internet, but have not been widely integrated into data management strategies for the grid. Doing so is likely to lead to useful insights into scalable replication approaches for grid environments. Replica management systems are similar in many respects to distributed file systems that support data redundancy. The main difference is that the management of file location information and meta-data is generally separate from the actual storage mechanisms.

A way of ensuring that computation and associated data are close together, proposed in [13], is to group compute and storage resources into *execution domains*, each of which is a set of resources that has high speed connections between them. An example of an execution domain is a set of servers within a single department, with local access to a large file store. Inter-domain access is generally slower, and therefore desirable to avoid. Computation is scheduled such that it resides in the same execution domain as the data it accesses. Data can also be dynamically replicated or migrated to other domains, depending on where it is needed by applications. A similar concept to execution domains is that of I/O communities [112], where groups of computation hosts share a common storage appliance. Computation and data is assigned to I/O communities based on data access needs of the jobs. Both of these rely on Condor's ClassAds mechanism [90], which allows resources and jobs to specify requirements. This allows available storage and computational resources to be chosen based on the needs of applications, in these cases specifically the requirement that computation be placed close to data.

While data scheduling, as used in this proposal, refers strictly to the movement of data, a closely related idea is that of scheduling computation based on knowledge about data locations. By placing jobs in such a manner, shorter execution times can be obtained due to faster access to the data. An example of this is the *Distributed Active Resource Architecture* (DARC) [71], which provides infra-structural support for *active data repositories*, which are essentially

virtual data sources where the data made available is a transformation of data stored on disk, and generated on-the-fly. A related concept is that of *active disks* [94], in which application code is executed on a processor built-in to the physical disk drive. The scheduling of computation and data together as part of the same process has received little attention, and work has been proposed to look at how replication strategies can be combined with traditional computation scheduling techniques [122]. The research proposed in Section 3 shares some of the ideas behind this approach.

2.5.4 Service Scheduling

Service Scheduling [5] is the problem of allocating requests for service access to specific hosts providing that service. Whenever an application attempts to access a service to perform some operation, and a number of hosts on the grid are capable of granting the request, a decision must be made about which host should be selected. Factors to be considered when making this decision include the computational cost of performing the operation, the amount of data that must be transferred as part of the request or response, external data sources that the service needs to access, and the load on the machines that provide the service. During the course of its lifetime, an application may access a single instance of the service on the one host, or multiple instances across different hosts.

When accessing a service, a program specifies a particular host it wants to connect to. This is done by supplying a network address, and obtaining a handle to an object which can then be used to invoke operations. The address is usually either hard-coded into the application or read from a configuration file. Once the connection has been established, it is maintained over a period of time, and used for multiple operation invocations. For example, accessing a remote object using Java's RMI [109] requires a client to specify a URL string referencing the object on a particular host, and returns a *stub* object referencing the service on the remote host, which can subsequently be used by a client for multiple operation invocations. A similar method is used in CORBA [82], where clients *bind* to an object on a remote host, and use the resulting stub object to invoke remote operations. The equivalent concept in the context of grid services [116] is that of the Grid Service Reference (GSR), which can be used to send requests to a remote service, and is obtained in a similar manner.

Instead of requiring a specific network address to be supplied, some systems provide support for *service discovery*. This allows a program to specify a particular service it wants, and then discover the set of available hosts that provide the service. Using this approach allows more flexibility, because the decision about which host to access a service on is postponed until runtime, where it can be made based on information that was not necessarily available when the program was written or compiled. A common way of performing service discovery is to consult a *registry*, which contains information about a set of services that have been registered with it. A client specifies certain search criteria, including an identifier corresponding to the interface it wants, and receives a list of matching services to which it is able to connect. The registry is commonly implemented as a directory service, such as MDS-2 [35], SDS [38], or UDDI [10]. Additional ways of enhancing this search capability include semantic annotations to services [105].

Another method of discovering services is for clients to broadcast requests across the entire network, hoping for a response from one or more hosts providing the service. An example of this is the discovery mechanism for lookup services in Jini [120], which is designed to allow devices to join or leave a network without configuration changes. While this approach removes the need to specify the location of a registry, the broadcast mechanism does not scale to large networks, and is therefore undesirable for grid environments. Peer-to-peer discovery services [12, 58] have been suggested as another approach which provides a higher level of scalability.

Once all available service instances have been found, and one has been selected for use, a client may begin submitting operation requests. However, after some period of time, it may be desirable to switch to a different instance of a service if the host initially selected fails or becomes heavily loaded; these features are known as *fail-over* and *load balancing* respectively. If the service instance being accessed is stateless, and switching to a different one will have no side-effects on the application's behaviour, then the application can choose a different instance to which it will submit further operation invocation requests. To do this, it is necessary to *reschedule* the service access by obtaining the latest list of available hosts providing the service, and performing selection again. The references within the application must then be updated to point to the location of the new service instance by *dynamically re-binding* the relevant stub or proxy object used to access the service [75].

Often, an *object request broker* (ORB) is used by applications to provide location transparency of services [100]. This allows a program to access an object and invoke methods without knowing which specific host the request will be sent to. At the implementation level, it will often be here that the scheduling decisions are implemented, by having the ORB contact the appropriate host when service requests are made by an application. Proxy objects used within the application typically interact with the ORB to provide this functionality.

From a scheduling perspective, changing an application's service reference to point to a different instance on another host is a similar concept to migrating data or tasks during execution. Viewed from a high-level, it is simply a component of the application that is being moved from one host on the grid to another. This makes sense when considering the work done by a service on behalf of the application to be part of the application itself, in the same sense that the code in a shared library can also be considered part of the application. The concept of scheduling components of a running application, namely data and computation as described above, can thus be extended to include services. Previous work has not considered the scheduling of service access in the same manner as other application components; this deficiency is addressed in the work proposed in Section 3.

2.6 Summary

A large amount of research to date has considered the problem of providing effective ways of utilising heterogeneous collections of computing resources in a grid environment. System and usage models vary widely, and many schemes have been demonstrated for allocating applications to resources in order to optimise the performance. There are three main types of scheduling that are performed to support this goal, namely those of *computation*, *data*, and *services*. However, these types of scheduling have been considered separate from one another, and attention has not been given to integrating the strategies used for such placement. Additionally, current approaches to computation scheduling are primarily focused toward sequential jobs, and even those that support parallel jobs rarely take into account the locality requirements and dependencies between tasks in the same process. The interaction between tasks has similar implications for scheduling strategies to that of data and service access, however no existing schedulers take this into consideration. It is these gaps which we propose to address with this project.

3 Proposed Research

The research described here proposes to investigate the gaps identified in the previous section by developing and evaluating mechanisms for performing scheduling of tasks, files, and services in a grid environment. While a major aim is to develop these concepts at a theoretical level, our investigation will also look at implementation in existing middleware.

3.1 Experimental Architecture

In order to contain the scope of the project, models for the grid environment and types of applications have been chosen. These consist of a set of assumptions about the types of systems being considered here. All research done in this project will specifically apply to systems which fit these models; to extend it beyond this, some of the approaches taken or algorithms used may need to be modified in order to satisfy additional requirements. The assumptions made here have been chosen to cover a wide range of grids that exist now and that are likely to be developed in the future. Note that the model described here does not prescribe any implementation details; the protocols, programming languages, operating systems, standards, user interfaces and other software or components are not specified. While particular technologies will be used for exploring these concepts, the proposed research is targeted at the theoretical level, in order to be applicable to a range of implementations.

3.1.1 Grid Model

A *grid* consists of a set of one or more *resources*, each of which is a single computer with some amount of memory and storage space. The machine may have either a single processor or multiple processors with access to shared memory, although from an external point of view this is irrelevant. Access to the facilities of the resource is provided through *middleware*, a software layer on top of the underlying operating system.

Some resources on the grid are capable of executing multiple tasks at a once, using time-slicing mechanisms provided by the operating system or intermediate runtime system, while others can execute only one task at a time. The local schedulers on each resource initiate tasks assigned to them immediately, without delays caused by job queuing systems waiting for other processing to complete, or reject the job if multitasking is not supported and the processor is busy. This prevents parallel applications from being delayed by having their tasks scheduled to run at different times.

Each task runs within a specific *execution environment*, which executes the sequence of instructions associated with the task. This can be done through various mechanisms such as running a binary executable natively on the processor, through a virtual machine, or in an interpreter. An execution environment also includes a set of interfaces to the operating system and shared libraries. A host may provide zero or more execution environments. Each task requires a specific execution environment, and can thus only be assigned to hosts which provide that particular environment.

A host may also provide zero or more *services*, each of which can be accessed through a messaging protocol which is standardised across the entire grid. Like tasks, these services are implemented in a specific execution environment provided by the host. A service may be accessed by tasks running either on the same host, or other hosts. The set of services provided by a host is fixed; it cannot be changed by a running application. In this model, any computational entity which can be sent to a remote host for execution is considered to fall under the definition of a task, as the mechanisms used to communicate with it are implementation details only.

In addition to the work assigned by a scheduler, each resource may be running one or more other tasks that are not visible to or under the control of the metascheduler. These typically consist of operating system or user processes run by the owner of the resource. This means that the processor load can vary according to the execution of these other tasks in ways that cannot be predicted by the metascheduler.

A resource may become unavailable for use at any point in time, due to various reasons such as network disconnection, operating system crash, middleware failure, or owner-initiated shutdown. It may then become available again at a later point in time. Any scheduler built for the grid must therefore be capable of handling changes in the set of available resources. When resources disappear, the failure should be reported to the end user of an application, or the application should be moved to or restarted on a different set of resources. When new resources appear, the scheduler should be able to make use of them.

Data that is accessed by applications is accessible from any point on the grid in a transparent manner. The details how how a data file is accessed are abstracted away from the application by the middleware, so that it does not matter

if the data is local or remote. Any transfer or remote access mechanisms necessary are handled by the middleware, either before or during execution of the application.

3.1.2 Application Model

A *process* is a running instance of a *program*, a sequence of instructions contained in a source or binary file. A process may be *sequential*, consisting of a single task, or *parallel*, consisting of multiple tasks. The number of tasks in a process may vary over the course of its lifetime, and the tasks may communicate and interact with one another through the use of message passing. Additionally, a task may access *services* running on the same or remote hosts; this is done using messages exchanged through an RPC protocol, and the computation performed by the service is done on behalf of the process. Each task in a process may also access one or more *data files*, residing either locally or remotely.

All tasks in the process require the same execution environment, and may run on any host in the grid that provides that environment. Tasks may access services regardless of which execution environment they are implemented in, since the interaction occurs only through standard, platform-independent communication mechanisms.

The time required to execute a process cannot be predicted. It is difficult enough for a user to obtain an accurate estimate of their application's execution time in a fairly controlled environment such as a parallel computer; in a grid environment, with many different factors affecting application performance, this becomes near impossible. No scheduling algorithm designed within the context of this model should rely on an application completing within a specific time frame.

3.2 A Unified Approach to Scheduling

As discussed in Section 2, there are a number of different types of scheduling that have been applied to grid and parallel computing, and many different approaches have been taken to the problem of doing this scheduling. These are summarised here.

Computation scheduling refers to the process of deciding where to launch programs, or parts of programs, on a collection of hosts. A program is submitted to be run as a job to the scheduler, which then uses information about the program and the available hosts on the grid to choose an appropriate set of hosts to execute the threads of a program, and then takes the necessary steps to get it running on those hosts. This process is typically done with the goal of minimising the execution time of the program.

Data scheduling refers to the problem of finding an appropriate place on the grid to store data that is to be used by applications. By placing data files close to the applications that use them, they can complete in a shorter period of time due to increased speed of access to the files, and a lower load will be placed on the network.

Service scheduling is the process of deciding which host, out of a set which provides a given service, will be used by an application. Choosing a host which is likely to execute the relevant operations on the service in the shortest period of time enables application performance to be optimised.

Each of these types of scheduling has similar requirements in terms of the information available and the decisions that need to be made. Because of these similarities, we propose a model of scheduling that operates on generic *schedulable entities*. A scheduling algorithm designed within this model would not be aware that it is dealing with tasks that are launched or migrated, data files that are copied from one location to another, or which machines are chosen for a program to access a particular service. It would operate only on the higher-level description of these entities which abstracts away from the details of what role they actually play in the grid environment.

This high-level view simplifies the problem of making scheduling decisions, and allows us to focus on it purely from the algorithmic point of view, without having to consider implementation details of the grid, or specific programming or infrastructure technologies. Instead, these can be considered separately, with mechanisms constructed to provide input to the scheduler based on knowledge of the actual components of grid applications and resources, and take the resulting schedule and translate it into specific actions to be performed on the grid, such as launching tasks or copying data files.

Scheduling then becomes the conceptually much simpler problem of deciding which entities to place on which resources at particular points in time. Figure 3 shows an example representation of a system containing several entities and resources. The type of each of these entities has been deliberately omitted from the diagram, to emphasise its irrelevance to the scheduler. The information given to the scheduling algorithm consists of the set of entities, the set of resources, and additional, non-type-specific information about the entities and resources and the relationships between them. Examples of this might include a connection between two entities corresponding to communicating tasks, a property of an entity representing the computational and storage costs of placing it on a resource, or a link between an

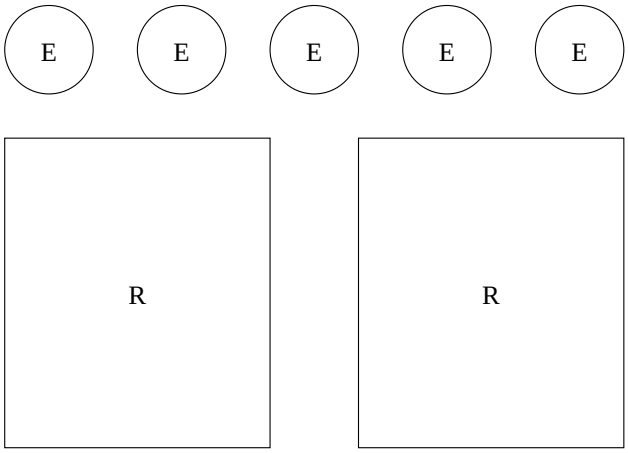


Figure 3: A set of entities and resources as viewed by the scheduler

entity and a resource indicating that this is where it currently resides. The *schedule*, which is the result of executing the scheduling algorithm, is a complete set of assignments of entities to resources.

Once the entity-resource assignments have been obtained from the scheduler, a separate component of the system translates this into a set of actions that implement the decisions. The new schedule is compared with the previous one to determine what changes have occurred; the actions taken depend on the differences between the two. In the case of new entities that were not present in the previous schedule because they have been appeared between executions of the algorithm, new tasks will be launched, data files created, or service access initiated. For entities that have moved since the previous schedule, tasks will be migrated, or data files moved, or service accesses redirected to a different host. The implementation details of how these changes are implemented is dependent upon the nature of the grid and the application. Some features, such as task or data migration, may not be supported in particular environments - this information would be given to the scheduling algorithm prior to its execution, so that the relevant restrictions can be placed on the changes that are made to certain entities.

Separating the scheduling of entities and the implementation of the resulting decisions in this manner provides a clean structure for developing scheduling mechanisms. It allows those developed for one type of grid or execution environment to be adapted to others, and simplifies the scheduling algorithms by reducing the amount of information they have to deal with. We see this as a more effective way of performing scheduling than just considering one type of entity, as all components of an application can be scheduled to suit the grid environment rather than just a subset of them.

3.3 Graph-based Program Representation

Any running program consisting of multiple entities may be represented as a graph. Each node in the graph represents a particular entity, whether it is an executing thread, a data file that is being read from or written to, or a service which is being accessed by a particular thread. The edges in the graph represent the relationships between the entities, such as communicating tasks or open file handles. This graph will be used as the central data structure that the scheduler operates on.

Figure 4 shows an example of a parallel program represented as a graph. This program contains four tasks, and also accesses two data files and a single service. Task 1 takes input from the user in the form of command-line parameters, and then interacts with tasks 2 and 3, which retrieve information by making calls to Service A and reading data from File X. Both of them process this information and then pass the result to Task 4, which performs some further processing and stores its output in File Y.

When scheduling a program, it is desirable to place entities which transfer a lot of data between each other close together, and entities which perform a lot of processing on powerful machines. For example, suppose Task 3 reads a large amount of data from File X, and then performs a computationally inexpensive search operation to extract a single value. It then passes it to Task 4, which, based on this value and the information received from Task 2, performs a highly intensive processing operation that requires a lot of CPU time. It would make sense to place Task 3 and the

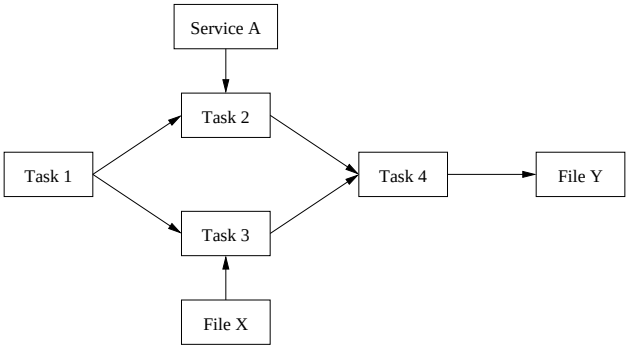


Figure 4: Graph representation of a program

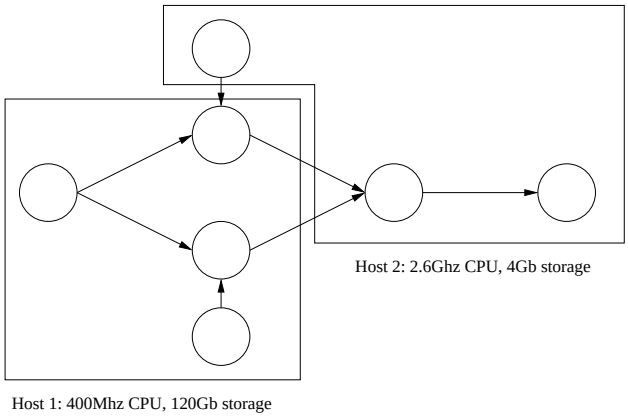


Figure 5: Process network representation and host assignment

file it reads from on the same host, so that the file can be accessed efficiently from the local disk instead of being transferred across the network, and to place Task 4 on a much more powerful host, transferring only the single value that it needs. Additionally, if the operations invoked on Service A required a lot of computation to be performed, but only returned small amounts of data, then it would be better to access an implementation of the service residing on a powerful, lightly loaded host, even if an inferior one with a faster network connection was available.

At a more abstract level, such graphs can be represented as *process networks* [64], which consist of a series of *nodes* connected together by *communication channels*. Each node in the network takes input on one or more channels, and sends output along one or more other channels. Although only unidirectional channels are permitted, cycles are allowed, so if a pair of nodes needs to send data in both directions then separate input and output channels can be constructed. In our model of schedulable entities, each is equivalent to a node in a process network. Tasks input and output data from each other through a message-passing system, from services through an RPC protocol, and from files through operating system APIs or I/O libraries.

To the scheduler, services and tasks are very similar, because they both incur a computation cost when assigned to a particular host, whereas data does not. A data file is like a task which consumes no CPU cycles but just sends and receives messages when read from or written to. Similarly, costs are associated with the links between entities, based on the amount of data transferred from a file, or sent as messages between tasks or services. These costs, as well as the capabilities of individual hosts, can be taken into account by the scheduler to make optimal placement decisions. Mechanisms for determining these costs will be an aspect of our research. In some cases this information may be available from the static program structure or information supplied by the user, and in others it could be measured dynamically by observing program behaviour during execution.

Figure 5 shows a process network corresponding to the program, and the hosts that specific nodes have been assigned to. Tasks 1, 2, and 3 have been placed along with File X onto a host with only moderate computational

power but a large amount of storage space. Task 4 and the service have both been placed on a more powerful host; the resulting output file resides there as well. Given these two example hosts, this allocation satisfies the requirements mentioned above in regards to efficient access to data files, and placement of computationally expensive tasks on powerful processors. Of course, there are many situations in which these goals may conflict, such as when expensive computation needs to be performed on large amounts of data, and hosts which satisfy both requirements are not available. In this case, trade-offs have to be made, and the scheduling algorithm must select the choice that it believes will lead to the best application performance.

Process networks provide a useful model for our experiments, and a significant amount of past work relating to these has been done which can be leveraged for our purposes. In particular, we will utilise PAGIS [124], an implementation of process networks designed for grid computing environments. PAGIS provides an architecture for representing and executing process networks, and enables distribution of the networks across multiple grid resources. The current implementation supports tasks written as Java classes, and allows them to be launched on and migrated to other nodes running the PAGIS environment, which integrates with the Globus toolkit. A graphical environment can be used to construct process networks and monitor their execution.

3.4 Aim

In this project, we propose to investigate the issues described above, and look at a range of different scheduling algorithms for placing the tasks of a parallel program, the data files they use and the services they access on grid resources. Our focus is on the conceptual level of scheduling, with the aim that the results of our research can be applied to a range of different parallel, distributed systems. We will investigate the algorithms within the context of multiple implementations, so that they can be tested in a range of different configurations.

3.5 Method

The research will be split into multiple phases, starting off with consideration of a specific grid execution environment, and then expanding out to a wider range of execution models and grid systems. The phases will be as follows:

1. Investigation of scheduling algorithms within the PAGIS application environment.

   The existing, simplified node placement strategies used in PAGIS will be extended with more advanced scheduling algorithms. These will take into account information about the process network and the connections between each of the nodes, in order to create appropriate schedules as described above.

   Because PAGIS provides a relatively controllable execution environment, extra functionality can be added to the execution system to monitor performance in different configurations. This will allow for experiments to be performed on a set of machines, and data collected to measure the performance of different scheduling strategies with a range of programs.

   The goal behind initially working with PAGIS is to get hands-on experience implementing scheduling in a real system. The work done in this part of the project will provide a useful basis for looking at scheduling in the more general case and application of the concepts to other systems later in the project.

2. Data scheduling

   Once the scheduling architecture from the previous step has been implemented, it will be extended to support the scheduling of file placement. This will be achieved by adding an additional type of process network node to PAGIS which supports reading or writing of data in a file. Input and output channels will be used to perform sequential reading or writing, so that it can be used within the existing architecture in the same manner as a node which produces or receives data. The main difference with file nodes is that when the node itself is moved to a different host, the data file goes with it. The mechanisms used for specifying a file will include support for files on remote machines running PAGIS, such that a file node will start on the host that contains the file, which may be different from the host that initially contains all the other nodes. Migration can be prevented for file nodes if the file is required to stay at its original location.

3. Service scheduling

   In a similar manner to the implementation of file access, an additional type of process network node will be added to support access to grid services. When one of these nodes is added to a process network, the user will

have to specify information about the service interface to be accessed, the operation to be performed, and what input channels tie to what parameters. The node will have a single output channel representing the return value of the operation. However, the specific host providing the service will not be specified - instead, information must be provided which the scheduler can use to find a list of hosts providing the service, and then decide which host to use in the same manner as it decides which host to execute a task or transfer a file to. This service access will thus be performed in the same way as that of tasks and files, but the implementation mechanisms will be specific to the use of services. As with the previous two phases, experiments will be performed to measure the performance of various scheduling algorithms in this situation.

4. Investigation of scheduling in simulation environment

   An existing grid simulation environment will be used to test the scheduling algorithms from the previous stages on a larger scale. This will allow for testing these algorithms with larger numbers of processors than physically available, by virtue of being able to set up a simulated grid. This stage will include an investigation of existing simulators that are suitable for use.

   A range of different configurations, from large to small, will be tested, to see how well each of the algorithms scales.

5. Application of concepts to other middleware

   By this point we will have developed a solid understanding of the applicability of metascheduling to different types of entities, namely tasks, files and services. We will also have investigated a number of different scheduling algorithms which can be used for this purpose, and ways of abstracting the different types of schedulable entities away from the actual scheduling process. The next step is to apply these concepts to other types of parallel programming systems in grid environments. We will consider a number of different candidate systems, and determine which are the most interesting from the point of view of applying these concepts.

   The previously-developed scheduler will be modified so that it acts as a component which can be easily interfaced with other types of systems. This interface will then be used to integrate the scheduler with several different types of grid middleware. Experiments will be carried out in a range of different configurations to see how well the algorithms works in each system. This will be compared to the simulated results to determine what issues arise that in practice. Where significant performance differences are observed, these will be investigated and, as far as possible, corrected for.

3.6 Milestones

October 2004

- Research proposal completed

November 2004

- Summary and evaluation of PAGIS for purpose of this project
- Revision/extension of PAGIS to provide a foundation for future work

March 2005

- Basic scheduling architecture implemented in PAGIS
- Cost mechanism for process network nodes and channels implemented
- PAGIS extended to support file and service nodes

September 2005

- Several centralised scheduling algorithms implemented in PAGIS, and evaluated on a small test grid
- Existing grid simulation environments investigated, and one selected for use

- Centralised scheduling algorithms implemented and evaluated in simulator

March 2006

- Distributed scheduling algorithms implemented and evaluated in PAGIS
- Distributed scheduling algorithms implemented and evaluated in simulator
- Existing middleware chosen for application of scheduling concepts
- Scheduler from PAGIS made into a re-usable component

September 2006

- Scheduler interfaced with existing middleware
- Scheduling algorithms investigated in context of existing middleware
- Results from middleware compared to those from simulator

March 2007

- Thesis completed

References

[1] 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.

[2] Eytan Adar and Bernardo A. Huberman. Free riding on gnutella. Technical report, Xerox PARC, August 2000.

[3] Ammar H. Alhusaini, Viktor K. Prasanna, and C.S. Raghavendra. A unified resource scheduling framework for heterogeneous computing environments. In *Proceedings of the 8th IEEE Heterogeneous Computing Workshop*, April 1999.

[4] Bill Allcock, Joe Bester, John Bresnahan, Ann L. Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steven Tuecke. Data management and transfer in high performance computational grid environments. *Parallel Computing Journal*, 28(5):749–771, May 2002.

[5] Esam Alwagait and Shahram Ghandeharizadeh. A comparison of alternative web service allocation and scheduling policies. In *IEEE International Conference on Services Computing*, Shanghai, China, September 2004.

[6] Thomas E. Anderson, David E. Culler, and David A. Patterson. A case for networks of workstations: NOW. *IEEE Micro*, February 1995. http://now.cs.berkeley.edu.

[7] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. In *In Proceedings of the 15th Symposium on Operating Systems Principles*, pages 109–126, Copper Mountain Resort, Colorado, December 1995. ACM.

[8] The Apache Software Foundation. WebServices - Axis, 2004. http://ws.apache.org/axis/.

[9] Apple Computer. AppleTalk filing protocol version 2.1 and 2.2. AppleShare IP 6.0 Developers Kit, 1998.

[10] Ariba Inc., IBM Corp., and Microsoft Corp. Universal description, discovery and integration (UDDI) technical white paper, September 2000. http://www.uddi.org/.

[11] Mark Baker, Geoffrey Fox, and Hon Yau. Cluster computing review. Technical Report CRPC-TR95623, Center for Research on Parallel Computation, Rice University, November 1995.

[12] Farnoush Banaei-Kashani, Ching-Chien Chen, and Cyrus Shahabi. WSPDS: Web services peer-to-peer discovery serivce. In *The International Symposium on Web Services and Applications (ISWS'04)*, June 2004.

[13] Jim Basney, Miron Livny, and Paolo Mazzanti. Utilizing widely distributed computational resources efficiently with execution domains. *Computer Physics Communications*, 2001.

[14] E. Bayeh. The WebSphere application server architecture and programming model. *IBM Systems Journal*, 37(3):336–364, 1998.

[15] Fran Berman, Richard Wolski, Silvia Figueira, Jennifer Schopf, and Gary Shao. Application-level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing 1996*, Pittsburgh, PA, November 1996.

[16] Guy Bernard and Michel Simatic. A decentralized and efficient algorithm for load sharing in networks of workstations. In *Proceedings of the EurOpen Spring '91 Conference*, Tromso, Norway, May 1991.

[17] Joseph Bester, Ian Foster, Carl Kesselman, Jean Tedesco, and Steven Tuecke. GASS: a data movement and access service for wide area computing systems. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 78–88, Atlanta, Georgia, United States, 1999. ACM Press.

[18] Tracy D. Braun, Howard Jay Siegel, Noah Beck, Ladislau Boloni, Muthucumaru Maheswaran, Albert I. Reuther, James P. Robertson, Mitchell D. Theys, and Bin Yao. A taxonomy for describing matching and scheduling heuristics for mixed-machine heterogeneous computing systems. In *IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 330–33, West Lafayette, IN, October 1998. (included in the Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems, 1998).

[19] Tim Brecht, Harjinder Sandhu, Meijuan Shan, and Jimmy Talbot. ParaWeb: Towards world-wide supercomputing. In *Proceedings of the 7th workshop on ACM SIGOPS European workshop*, pages 181–188, Connemara, Ireland, 1996. ACM Press.

[20] Rajkumar Buyya, editor. *High Performance Cluster Computing*, volume 1 and 2. Prentice Hall - PTR, NJ, USA, 1999.

[21] Henri Casanova and Jack Dongarra. Netsolve: A network server for solving computational science problems. Technical Report CS-95-313, University of Tennesee, November 1995.

[22] Henri Casanova, Graziano Obertelli, Francine Berman, and Rich Wolski. The AppLeS parameter sweep template: user-level middleware for the grid. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 60, Dallas, Texas, United States, 2000. IEEE Computer Society.

[23] C. Catlett, I. Foster, and W. Johnston. GGF structure. Global Grid Forum Document GFD.2, 2002.

[24] Ethan Cerami and Simon St.Laurent. *Web Services Essentials*. O'Reilly & Associates, Inc., 2002.

[25] Arjav J. Chakravarti, Gerald Baumgartner, and Mario Lauria. Application-specific scheduling for the organic grid. Technical Report OSU-CISRC-4/04-TR23, Dept. of Computer and Information Science, The Ohio State University, April 2004.

[26] Stephen Joel Chapin. *Scheduling support mechanisms for autonomous, heterogeneous, distributed systems*. PhD thesis, Purdue University, 1993.

[27] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven Tuecke. The data grid: Towards and architecture for the distributed management and analysis of large scientific data sets. *Journal of Network and Computer Applications*, 23(3):187–200, 2000.

[28] 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.

[29] Jonathan Chin and Peter V. Coveney. Towards tractable toolkits for the Grid: a plea for lightweight, usable middleware. UK e-Science Technical Report UKeS-2004-01, National e-Science Centre, February 2004.

[30] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *In Proceedings of the ICSI Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, 2000. International Computer Science Institute.

[31] Cluster Resources, Inc. TORQUE resource manager. http://supercluster.org/torque/.

[32] The Condor Project. Chirp homepage. http://www.cs.wisc.edu/condor/chirp/.

[33] Geoff Coulson, Gordon S. Blair, Michael Clarke, and Nikos Parlavantzas. The design of a configurable and reconfigurable middleware platform. *Distributed Computing*, 15(2):109–126, 2002.

[34] Karl Czajkowski, Don Ferguson, Ian Foster, Jeff Frey, Steve Graham, Tom Maguire, David Snelling, and Steve Tuecke. From open grid services infrastructure to WS-resource framework: Refactoring & evolution, March 2004. http://www.globus.org/wsrf/.

[35] Karl Czajkowski, Steven Fitzgerald, Ian Foster, and Carl Kesselman. Grid information services for distributed resource sharing. In *Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10)*. IEEE Press, August 2001.

[36] Karl Czajkowski, Ian Foster, Nick Karonis, Carl Kesselman, Stuart Martin, Warren Smith, and Steven Tuecke. A resource management architecture for metacomputing systems. *Lecture Notes in Computer Science*, 1459:62–82, 1998.

[37] Karl Czajkowski, Ian Foster, and Carl Kesselman. Resource coallocation in computational grids. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, August 1999.

[38] Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz. An architecture for a secure service discovery service. In *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 24–35, Seattle, Washington, United States, 1999. ACM Press.

[39] Alan Dickman and Peter Houston. *Designing Applications with MSMQ: Message Queuing for Developers*. Addison-Wesley Longman Publishing Co., Inc., 1998.

[40] Distributed Computing Department, Computational Research Division, Lawrence Berkeley National Laboratory. pyGridWare: Python web services resource framework. http://www-itg.lbl.gov/gtg/projects/pyGridWare/.

[41] Fred Douglis and John Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software: Practice and Experience*, 21(8):757–785, August 1991.

[42] Ian Foster (ed), Jeffrey Frey (ed), Steve Graham (ed), Steve Tuecke (ed), Karl Czajkowski, Don Ferguson, Frank Leymann, Martin Nally, Igor Sedukhin, David Snelling, Tony Storey, and Sanjiva Weerawarana. Modeling stateful resources with web services v. 1.1, March 2004. http://www.globus.org/wsrf/.

[43] Electronic Frontier Foundation. RSA code-breaking contest again won by distributed.net and electronic frontier foundation (EFF). Press Release, January 1999.

[44] Dror G. Feitelson. Job scheduling in multiprogrammed parallel systems. IBM Research Report RC 19970, August 1997.

[45] 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.

[46] Ian Foster, Carl Kesselman, Craig Lee, Bob Lindell, Klara Nahrstedt, and Alain Roy. A distributed resource management architecture that supports advance reservations and co-allocation. In *International Workshop on Quality of Service*, 1999.

[47] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. Open Grid Service Infrastructure WG, Global Grid Forum, June 2002.

[48] Richard F. Freund, Michael Gherrity, Stephen Ambrosius, Mark Campbell, Mike Halderman, Debra Hensgen, Elaine Keith, Taylor Kidd, Matt Kussow, John D. Lima, Francesca Mirabile, Lantz Moore, Brad Rust, and H. J. Siegel. Scheduling resources in multi-user, heterogeneous, computing environments with smartnet. In *Proceedings of the 7th IEEE Heterogeneous Computing Workshop (HCW '98)*, pages 184–199. IEEE Computer Society, March 1998.

[49] Joern Gehring and Thomas Preiss. Scheduling a metacomputer with uncooperative sub-schedulers. In *Proceedings of the 5th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP '99), in conjunction with the International Parallel and Distributed Processing Symposium (IPDPS '99)*, San Juan, Puerto Rico, April 1999.

[50] Joern Gehring and Alexander Reinefeld. MARS - a framework for minimizing the job execution time in a metacomputing environment. *Future Generation Computer Systems*, 12(1):87–99, 1996.

[51] Cecile Germain, Vincent Neri, Gille Fedak, and Franck Cappello. XtremWeb: building an experimental platform for global computing. In *Proceedings of the 1st IEEE/ACM International Workshop on Grid Computing (Grid 2000)*, Bangalore, India, December 2000. Springer Verlag Press, Germany.

[52] Nathaniel S. Good and Aaron Krekelberg. Usability and privacy: a study of Kazaa P2P file-sharing. Technical Report HPL-2002-163, HewlettPackard Labs, 2002.

[53] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 173–182, Montreal, Quebec, Canada, 1996. ACM Press.

[54] Andrew S. Grimshaw and Virgilio E. Vivas. FALCON: A distributed scheduler for MIMD architectures. In *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 149–163, Atlanta, GA, March 1991.

[55] Brian Hayes. Computing science - collective wisdom. *American Scientist*, March-April 1998.

[56] R.L. Henderson and D. Tweten. Portable batch system: Requirement specification. NAS technical report, NASA Ames Research Center, April 1995.

[57] Elisa Heymann, Miquel A. Senar, Emilio Luque, and Miron Livny. Adaptive scheduling for master-worker applications on the computational grid. In *Proceedings of the First IEEE/ACM International Workshop on Grid Computing (GRID 2000)*, Bangalore, India, December 2000.

[58] Wolfgang Hoschek. Peer-to-peer grid databases for web service discovery. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. Wiley Press, November 2002. (to appear).

[59] Wolfgang Hoschek, Javier Jaen-Martinez, Asad Samar, Heinz Stockinger, and Kurt Stockinger. Data management in an international data grid project. In *Proceedings of the 1st IEEE/ACM International Workshop on Grid Computing (Grid 2000)*, Bangalore, India, December 2000. Springer Verlag Press, Germany.

[60] Marty Humphrey, Glenn Wasson, Mark Morgan, and Norm Beekwilder. An early evaluation of WSRF and WS-notification via WSRF.NET. In *2004 Grid Computing Workshop (associated with Supercomputing 2004)*, Pittsburgh, PA, November 2004.

[61] IBM Corporation. Load leveler users guide, version 1.2, 1995.

[62] Eric J. Johnson. *The Complete Guide to Client/Server Computing*. Prentice Hall, January 2001.

[63] Michael B. Jones. Interposition agents: transparently interposing user code at the system interface. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 80–93, Asheville, North Carolina, United States, 1993. ACM Press.

[64] G. Kahn. The semantics of a simple language for parallel programming. In J. Rosenfeld, editor, *Proceedings of the International Federation for Information Processing Congress 74*, pages 471–475, Stockholm, Sweden, August 1974. North Holland Publishing Company.

[65] Rajkumar Kettimuthu, Vijay Subramani, Srividya Srinivasan, Thiagaraja Gopalasamy, and D. K. Panda P. Sadayappan. Selective preemption strategies for parallel job scheduling. In *2002 International Conference on Parallel Processing (ICPP'02)*, Vancouver, B.C., Canada, August 2002.

[66] Eric Korpela, Dan Werthimer, David Anderson, Jeff Cobb, and Matt Lebofsky. SETI@home - massively distributed computing for SETI. *Computing in Science & Enginering*, 3(1):78–83, 2001.

[67] Vipin Kumar, Ananth Y. Grama, and Vempaty Nageshwara Rao. Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Computing*, 22:60–79, 1994.

[68] Stefan M. Larson, Christopher D. Snow, Michael Shirts, and Vijay S. Pande. Folding@home and genome@home: Using distributed computing to tackle previously intractable problems in computational biology. In Richard Grant, editor, *Computational Genomics*. Horizon Press, 2002. (to appear).

[69] C. Lau and A. Ryman. Developing XML web services with websphere. *IBM System Journal*, 41(2), 2002.

[70] P. Leach and D. Perry. CIFS: A common internet file system. *Microsoft Interactive Developer*, November 1996.

[71] Jimmy Lee. Active data repositories for computational grid applications. Technical Report DHPC-133, University of Adelaide, October 2002.

[72] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997.

[73] M.L. Liu. *Distributed Computing: Principles and Applications*. Addison Wesley, 1st edition, June 2003.

[74] Miron Livny. *The Study of Load Balancing Algorithms for Decentralized Distributed Processing Systems*. PhD thesis, Weizmann Institute of Science, 1983.

[75] Feng Lu and Kris Bubendorfer. A RMI protocol for aglets. In *Proceedings of the 27th conference on Australasian computer science*, pages 249–253, Dunedin, New Zealand, 2004. Australian Computer Society, Inc.

[76] 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.

[77] Drew Major, Greg Minshall, and Kyle Powell. An overview of the NetWare operating system. In *Proceedings of the 1994 Winter USENIX Technical Conference*, pages 355–372, San Francisco, CA, January 1994.

[78] Ahuva W. Mu'alem and Dror G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543, 2001.

[79] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[80] Noam Nisan, Shmulik London, Ori Regev, and Noam Camiel. Globally distributed computation over the internet: The POPCORN project. In *International Conference on Distributed Computing Systems (ICDCS'98)*, Amsterdam, The Netherlands, May 1998. IEEE CS Press.

[81] Object Management Group. IDL C++ language mapping specification. Object Management Group (OMG), Technical Paper 94-09-14, 1994.

[82] Object Management Group. The common object request broker: Architecture and specification (CORBA), revision 2.0, 1995.

[83] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Mark Greenwood, Carole Goble, Anil Wipat, Peter Li, and Tim Carver. Delivering web service coordination capability to users. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 438–439, New York, NY, USA, 2004. ACM Press. http://taverna.sf.net.

[84] Oracle Corporation. Oracle application server, 2004. http://www.oracle.com/appserver/index.html.

[85] Oracle Corporation. Oracle BPEL process manager, 2004. http://www.oracle.com/technology/bpel/.

[86] Andy Oram, editor. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. OReilly Press, USA, 2001.

[87] Sang-Min Park and Jai-Hoon Kim. Chameleon: A resource scheduler in a data grid environment. In *Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 258. IEEE Computer Society, 2003.

[88] Dave Pearson. Data requirements for the grid: Scoping study report. Presented at GGF4, Toronto, 2002.

[89] Platform Computing Corporation. LSF user's and administrator's guide, November 1993.

[90] Rajesh Raman, Miron Livny, and Marvin Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, Chicago, IL, July 1998.

[91] Kavitha Ranganathan and Ian Foster. Decoupling computation and data scheduling in distributed data-intensive applications. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, page 352, Edinburgh, Scotland, July 2002. IEEE Computer Society.

[92] Kavitha Ranganathan and Ian T. Foster. Identifying dynamic replication strategies for a high-performance data grid. In *Proceedings of the Second International Workshop on Grid Computing*, pages 75–86. Springer-Verlag, 2001.

[93] L. S. Revor. NQS users guide. Computing and Telecommunications Division, Argonne National Laboratory, September 1992.

[94] Erik Riedel. *Active disks: remote execution for network-attached storage*. PhD thesis, Carnegie Mellon University, November 1999.

[95] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 188–201, Banff, Alberta, Canada, 2001. ACM Press.

[96] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun network filesystem. In *Summer Usenix Conference Proceedings*, pages 119–130, June 1985.

[97] Mitsuhisa Sato, Hidemoto Nakada, Satoshi Sekiguchi, Satoshi Matsuoka, Umpei Nagashima, and Hiromitsu Takagi. Ninf: A network based information library for a global world-wide computing infrastructure. In *Proceedings of High-Performance Computing and Networking '97*, pages 491–502, Vienna, April 1997.

[98] M. Satyanarayanan. On the influence of scale in a distributed system. In *Proceedings of the 10th international conference on Software engineering*, pages 10–18, Singapore, April 1988.

[99] M. Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[100] Douglas C. Schmidt and Chris Cleeland. Applying patterns to develop extensible ORB middleware. *IEEE Communications Magazine Special Issue on Design Patterns*, April 1999.

[101] Jennifer M. Schopf. Ten actions when superscheduling. Global Grid Forum Document GFD.04, 2003.

[102] Roger Sessions. *COM and DCOM: Microsoft's vision for distributed objects*. John Wiley & Sons, Inc., 1998.

[103] John F. Shoch and Jon A. Hupp. The "worm" programs – early experience with a distributed computation. *Communications of the ACM*, 25(3):172–180, 1982.

[104] Vladimir Silva. The master managed job factory service (MMJFS). *IBM developerWorks*, May 2004.

[105] Kaarthik Sivashanmugam, Kunal Verma, Amit Sheth, and John Miller. Adding semantics to web services standards. In *The 2003 International Conference on Web Services (ICWS'03)*, June 2003.

[106] R. Srinivasan. RPC: Remote procedure call protocol specification version 2. Internet RFC 1831, August 1995.

[107] Jaspal Subhlok, Peter Lieu, and Bruce Lowekamp. Automatic node selection for high performance applications on networks. In *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 163–172. ACM Press, 1999.

[108] Sun Microsystems. Java (TM) 2 platform, standard edition, v 1.4.2 API specification. http://java.sun.com/j2se/1.4.2/docs/api/index.html.

[109] Sun Microsystems. Java remote method invocation specification, October 1998.

[110] Andrew S. Tanenbaum and Robbert van Renesse. Distributed operating systems. *ACM Computing Surveys*, 17(4):419–470, December 1985.

[111] Kenjiro Taura and Andrew Chien. A heuristic algorithm for mapping communicating tasks on heterogeneous resources. In *Heterogeneous Computing Workshop*, pages 102–115, Cancun, Mexico, May 2000.

[112] Douglas Thain, John Bent, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, and Miron Livny. Gathering at the well: Creating communities for grid I/O. In *Proceedings of Supercomputing 2001*, Denver, Colorado, November 2001.

[113] Douglas Thain and Miron Livny. Bypass: A tool for building split execution systems. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing*, pages 79–85, Pittsburgh, PA, August 2000.

[114] Douglas Thain and Miron Livny. Parrot: Transparent user-level middleware for data-intensive computing. In *Workshop on Adaptive Grid Middleware*, New Orleans, Louisiana, September 2003.

[115] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The condor experience. *Concurrency and Computation: Practice and Experience*, 2004.

[116] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maguire, T. Sandholm, P. Vanderbilt, and D. Snelling. Open grid services infrastructure (OGSI) version 1.0. Global Grid Forum Draft Recommendation, 6/27/2003.

[117] Sathish S. Vadhiyar and Jack J. Dongarra. A metascheduler for the grid. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, page 343, Edinburgh, Scotland, July 2002. IEEE Computer Society.

[118] Sudharshan Vazhkudai, Steven Tuecke, and Ian Foster. Replica selection in the globus data grid. In *Proceedings of the First IEEE/ACM International Conference on Cluster Computing and the Grid (CCGRID 2001)*, pages 106–113. IEEE Computer Society Press, May 2001.

[119] Gregor von Laszewski, Ian Foster, Jarek Gawor, Warren Smith, and Steve Tuecke. CoG kits: A bridge between commodity distributed computing and high-performance grids. In *ACM 2000 Java Grande Conference*, pages 97–106, San Francisco, CA, June 2000.

[120] Jim Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82, 1999.

[121] Carl A. Waldspurger, Tad Hogg, Bernardo A. Huberman, Jeffrey O. Kephart, and Scott Stornetta. Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering*, February 1992.

[122] Daniel Walton. Replication strategies and computation scheduling in data grids. Research proposal, The University of Adelaide, October 2003.

[123] Glenn Wasson, Norm Beekwilder, Mark Morgan, and Marty Humphrey. OGSI.NET: OGSI-compliance on the .NET framework. In *4th IEEE/ACM International Symposium on Cluster Computing and the Grid (ccGrid 2004)*, Chicago, Illinois, April 2004.

[124] Darren Webb and Andrew L. Wendelborn. The PAGIS grid application environment. Submitted to 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid'03), Tokyo, Japan, 12-15 May, 2003.

[125] Dave Winer. XML-RPC specification, October 1999. http://www.xmlrpc.com/spec.

[126] World Wide Web Consortium (W3C). Web services description language (WSDL) 1.1, 2001. http://www.w3.org/TR/wsdl.

[127] World Wide Web Consortium (W3C). Simple object access protocol (SOAP) version 1.2, 2003. http://www.w3.org/TR/soap.

[128] Derek Wright. Cheap cycles from the desktop to the dedicated cluster: combining opportunistic and dedicated scheduling with Condor. In *Conference on Linux Clusters: The HPC Revolution*, Champaign - Urbana, IL, June 2001.

[129] Songnian Zhou, Xiaohu Zheng, Jingwen Wang, and Pierre Delisle. Utopia: a load-sharing facility for large heterogeneous distributed computing systems. *Software – Practice and Expeience*, 23(2):1305–1336, December 1993.

[130] Dmitry Zotkin and Peter J. Keleher. Job-length estimation and performance in backfilling schedulers. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, August 1999.