Zone Based Scheduling: A Framework for Scalable Scheduling of SPMD parallel programs on the Grid

Sandeep Prabhakar

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science

June 16th, 2003
Blacksburg, Virginia

Keywords: Grid Scheduling, Application Representation and Mapping, Grid Computing, MPICH-G2, Globus.

Zone Based Scheduling: A Framework for Scalable Scheduling of SPMD parallel programs on the Grid

Sandeep Prabhakar

Abstract

Grid computing is a field of research that combines many computers from distant locations to form one large computing resource. In order to be able to make use of the full potential of such a system there is a need to effectively manage resources on the Grid. There are numerous scheduling systems to perform this management for clusters of computers and a few scheduling systems for the Grid. These systems try for optimality (or close to optimality) with the goals of obtaining good throughput and minimizing job completion time.

In this research, we examine issues that we believe have not been tackled in schedulers for the Grid. These issues revolve around the problem of coordinating resources belonging to separate administrative domains and scheduling in this context. In order for grid computing's vision of virtual organizations to be realized to its fullest extent, there is a need to implement and test schedulers that find resources and schedule tasks on them in a manner that is transparent to the user. These resources might be on a different administrative domain altogether and obtaining either resource or user account information on those resources might be difficult. Also, each organization might require their own policies and mechanisms to be enforced. Hence having a centralized scheduler is not feasible due to the pragmatics of the Grid.

There are two basic aims to this thesis. The first aim is to design and implement a framework that takes administrative concerns into consideration during scheduling. The aim of the framework is to provide a lightweight, extensible, secure and scalable architecture under which multiple scheduling algorithms can be implemented. Second, we evaluate two prototypical of scheduling algorithms in the context of this framework. Scheduling algorithms are diverse and the applications are varied. Thus no single algorithm can obtain a good mapping for every application. We believe that different scheduling algorithms will be necessary to schedule different types of applications. In order to facilitate development of such algorithms, a framework in which it is easy to integrate other scheduling algorithms is necessary. The framework developed in this project is designed for such extensibility.

Acknowledgements

Thank you: **Dr. Calvin J. Ribbens**, for guidance in real life parallel applications and parallel programming, and for the caring and support that was provided throughout my progress. **Dr. Eunice Santos**, for help in solving problems related to scheduling and analyzing the results. **Dr. Srinidhi Varadarajan**, for his simulator on which the experiments were performed and for solving issues related to coding. **Prachi Bora**, for being my friend - someone with whom I can share any problem and someone who was there for me when the going got tough. **Muthukumar Thirunavukkarasu**, for his help with the simulator. The **MPICH mailing lists** for solving problems related to internal details of MPI implementation that was not documented.

Table of Contents

List Of Figures

List of Tables

List Of Symbols

z	=	Number of Zones
P	=	Total Number of Processors
T	=	Total Number of Tasks
P_i	=	Number of Processors on Zone i

Chapter 1. Introduction

Grid computing has become popular in the past few years as an effective means to couple large, heterogeneous sets of resources from multiple organizations by means of networks [46]. The Grid provides a computing infrastructure that is capable of running parallel applications that are too large to fit on a single cluster or parallel machine. Grid computing may prove to be a commercially viable alternative to building larger and larger machines for the same reason clusters became viable – the Grid is built on cheap off-the-shelf components. Grids are becoming popular mainly due to progress in networking technology, since network speeds have been increasing at a faster rate compared to processor speed, so that network bandwidth is becoming less and less of a bottleneck in the execution of a parallel program.

However, with all the advantages of coupling resources owned by multiple institutions, there are many challenges and problems that need to be tackled to make effective use of grid resources. Issues like resource management, reservation of resources, checkpointing and migration, and runtime algorithm selection are important to allow for good utilization of computing resources. Other pragmatic aspects like security of resources and privacy of resource structure need to be taken into account as well. Finally, the algorithms and infrastructure of the Grid, like the Internet, need to scale to provide for effective coupling of a large number of resources.

1.1 Resource Management for the Grid

Managing grid resources has been a central topic of interest to the grid community for many years [51]. In grid environments it is important to be able to find appropriate available resources from a large pool of resources. Once these resources have been identified, scheduling decisions must be made, i.e., tasks must be assigned to resources. This "mapping" problem has been studied by many researchers, and it is especially difficult in the dynamic and heterogeneous context of the Grid [50]. Wide-area scheduling systems have been studied in grid environments as a means to manage resources [2, 37, 38]. These systems are sometimes called resource brokers since they help the application to find the resources that the application requires.

There are many scheduling systems available for a single cluster of machines or a set of resources owned by a single organization. These scheduling systems have differing aims. Some of these systems aim to have high utilization of the machines and high throughput for multiple applications, while other systems try to minimize job completion time for a single application. Thus, there are many goals that a scheduler might be expected to satisfy, some of which cannot be achieved simultaneously. One such set of goals that is especially prominent in the Grid, and that cannot be achieved simultaneously, is respect for administrative boundaries and optimality of schedules. This research is an effort to identify and provide a solution for grid scheduling despite the constraints imposed by separate administrative domains.

1.2 Aims of Thesis

The research conducted in this project has the following two goals.

1. Development of a scheduling framework

   Scheduling of applications on grid environments is difficult due to the scale and diversity of the resources. In order for scheduling to be manageable, the user should not be burdened with the effort of collecting all the information required to schedule programs and then performing the mapping. What is needed is a scheduling framework that eases the burden on the user and transparently schedules applications. One of the aims of this research is to develop a scheduling framework that allows the user to easily use schedulers in meta-computing systems like the Grid. An extensible framework is designed so that different scheduling algorithms can work under this framework. Also, if grid-scheduling systems are to become popular, they will need to take into account the need for different organizations to have complete control of their resources, including setting policies for access to resources and sharing of information about resources. The framework developed in this project is ***zone-based***. Each zone represents an organization with its own scheduler. Each zone's scheduler is free to implement its own scheduling algorithms and enforce its own policies. Thus the zone-based scheduling framework handles the pragmatics of the Grid.

2. Evaluate scheduling algorithms in the context of this framework

   We evaluate a few scheduling algorithms in the context of our system. This provides insights into how the scheduling algorithms work in the context of our framework. The purpose of implementing and evaluating scheduling algorithms in our framework is not to provide the optimal schedule. Instead, we provide an implementation of a few scheduling algorithms in our framework and evaluate techniques that might allow further development of these scheduling algorithms.

1.3 Taxonomy of Grid Schedulers

Grid schedulers can be classified based on different criteria. A list of criteria used to classify schedulers is as follows:

- **Scheduling Methodology or Algorithm: Heuristic vs. Exhaustive**

  Since optimal scheduling is an NP hard problem, except for trivial cases, most existing scheduling algorithms use heuristics to perform task mapping.

- **Manner of Scheduling: Centralized vs. Distributed**

  Scheduling frameworks can be classified as being centralized or distributed. In centralized schedulers, the application's performance model and resource model are gathered at a single place and the mapping is performed at that location. In a distributed scheduler, there are many cooperative scheduling entities that collectively do the mapping in a peer-to-peer or a hierarchical manner.

- **Static, Dynamic or Adaptive:**

  Static schedulers use a predefined analytic model of the machines and networks, and then perform scheduling based on this model. Static schedulers do not take into account the effects of currently executing applications and the currently available processing power and bandwidth to an application. Dynamic schedulers take these variable factors about the resources into account while scheduling. Adaptive schedulers change the schedules or placement of tasks if after assignment of tasks, it is found that the conditions present on the resources have changed. Rescheduling systems that checkpoint currently executing tasks and reassign them to other resources are an example of such adaptive schedulers.

- **Generality of Scheduling Mechanism:**

  Schedulers can be specific for each application or can be generically used to schedule any application. Thus on one side we have application specific schedulers that are designed for a single application taking into account problem specific parameters. On the other hand we have generic schedulers that would work for any application. A hybrid between these two classes of schedulers can also be defined. A hybrid scheduler can schedule any application but allows performance information specific to the application to be provided.

The zone-based scheduler developed in this project can be categorized as a heuristic, distributed, dynamic and hybrid scheduler.

1.4 Organization

The rest of this thesis is organized as follows. Chapter 2 gives an overview of related work in the area. Based on the two goals of this research it provides information about scheduling frameworks and about scheduling algorithms and techniques. It also provides an introduction to the tools used to build the zone based scheduling (ZBS) framework. Chapter 3 describes the design and implementation of the ZBS framework. Chapter 4 describes different scheduling algorithms evaluated in the context of the framework. Chapter 5 provides the results obtained from the evaluation of the scheduling algorithms described in Chapter 4. Chapter 6 concludes with a summary of contributions and directions for future research in the area.

Chapter 2. Survey of Related Literature

The zone-based scheduler designed and implemented in this thesis uses several existing tools and technologies. This chapter gives an introduction to the various software components used in the scheduling system, including resource information gathering frameworks, application performance monitoring tools and web service frameworks. Also, the zone based scheduling system follows a different architecture compared to existing systems. A description of some of the previous efforts to schedule an application in grid environments is provided. The most significant generic grid application scheduler, the GrADS scheduler [2], is described. Different formulations of the scheduling problem are explored and algorithms to solve these scheduling problems are examined.

2.1 Execution Environments

The zone-based scheduler designed and implemented in this project is targeted for parallel applications. More specifically, it is assumed that the parallel application uses the same code base with different potions of data assigned to each process, i.e., it follows a single program multiple data (SPMD) style of program execution. All processes in the parallel program use message passing between the processes as a means of communicating information. Although our design does not require any particular grid or message passing middleware, our implementation is based on the following de facto standards for running message passing SPMD programs on the grid.

Globus toolkit [9] provides an infrastructure to execute programs on the grid. The Globus toolkit is based on open standards and provides resource management, information management and data management functions. Globus is used to provide the job startup mechanisms for the parallel program. Globus also takes care of the security issues of starting up jobs on remote resources by means of authenticating with the remote host using the digital certificate provided by the user.

MPICH-G2 [8]**,** the implementation of the MPI message passing standard for use with the Globus toolkit, is used as the library that does communication between the processes. MPICH-G2 makes communication libraries effective on multiple architectures. Point-to-point communication and collective communication operations are efficiently supported in MPICH-G2.

2.2 Resource Information Gathering Frameworks

In order to be able to make informed decisions about which machines to place processes on, the current load at the compute nodes is required. In addition, since the target applications are parallel programs, the latency and bandwidth between these nodes is also of interest to the scheduler. The infrastructure used to obtain this information is described as follows.

MDS (Meta-computing Directory Service) [10] provides an interface by which information about resources can be published by the resource and queried by the user. For example, clients can query the MDS to receive information about processor load, processor architecture, memory available, file system information and network interfaces. The MDS is a part of the Globus toolkit and is an implementation of the LDAP specification. The LDAP data model represents information as a set of objects organized in a hierarchical namespace. This namespace is called the directory information tree (DIT) with each entry in it being uniquely identified in the X.500 representation [16].

NWS (Network Weather Service) [11] is a network monitoring and forecasting software. NWS runs sensors on each machine. These sensors monitor the CPU load and network latency/bandwidth and generate forecasts for those parameters for the near future. The information provided by NWS is reported to the MDS service by means of an information provider that is part of MDS (the information provider is called *nwsip*).

The zone-based scheduler is the consumer of the information stored in MDS. It does so by contacting an LDAP server and querying for a portion of the directory information tree. Access to MDS programmatically is made possible by means of the COGkit [17] API, which provides a library that allows for easy querying. The zone-based scheduler makes decisions as to where to schedule processes based on processor and network properties forecast by NWS for the duration of the application's expected execution.

2.3 Grid Schedulers

Scheduling applications on the grid requires both frameworks to handle the complexities of a heterogeneous environment and algorithms that actually perform the mapping of tasks to resources. This section examines existing architectures and frameworks to perform grid scheduling. The next section provides details of algorithms used to perform scheduling. The zone-based scheduler provides an alternative design for a scheduling framework, and one of the goals of this research is to evaluate scheduling algorithms described in the next section as they perform in the context of this framework.

Current grid schedulers can be classified under three categories. In the first category there are schedulers that exist for individual applications. In this type of scheduler, the scheduler is an agent acting on behalf of the user or application. This type of scheduler is very application specific; and there is one of these scheduling agents for every instance of every application to be run. An example of a scheduler in this category is the AppLeS [3, 18, 53] scheduler. In the second category there are the meta-schedulers that take many applications and try to resolve the contention effects between applications. An example of a scheduler in this category is the GrADS meta-scheduler [5]. In the third category of schedulers, there are schedulers at each resource, where a resource can be anything from a single SMP or cluster to an enterprise-wide grid. Schedulers in this third class are similar to meta-schedulers but are less sophisticated from the application's point of view in that they have no information of the performance model of the application. Examples of such schedulers include PBS [31], Load Sharing Facility (LSF) [41], Load Leveler [48] and MAUI [32]. In this section efforts to develop schedulers and meta-schedulers for grid computing environments are described.

Existing efforts to develop schedulers have focused on a centralized approach in that the scheduling decision takes place at a single place. In the case of the application specific scheduler and the GrADS scheduler the decision is made per application and in the case of the resource level schedulers the decision is made per resource. Thus, when scheduling, complete information about both the application and resources is gathered at that one place where the scheduling decision is being made. One of the goals of our research is to remove this reliance on a single scheduling point, so that scheduling can be scaled up to larger and larger grids.

The AppLeS project [3, 18, 53] maps tasks to resources and evaluates the mapping based on how such a mapping will affect the application. The schedulers developed by the AppLeS group are specific to each application. Thus when an application is written, the parameters in the application that affect its performance are quantified and a very specific scheduler written for that application. The components of the AppLeS project include a resource selector which filters resources, a planner which generates a schedule, a performance estimator which predicts the performance of the application based on performance parameters extracted from the application, and an actuator which implements the best schedule.

The AppLeS scheduler, while found to be effective, requires a substantial effort from the application developer due to the need to implement a performance model for every new application [42]. To improve this situation, the GrADS project [4, 6] is developing a modular scheduler [2] that allows plugging of components into an overall scheduling framework. Essentially, it provides a module to specify the problem parameters and a mapper module. These modules are given to the framework, and the framework handles the scheduling based on the user specified criteria. The framework is extensible in that to allow a new application to use this scheduler all that would be required is to write these modules and the framework would take care of actually implementing the rules specified by the user and evaluating the performance based on the criteria specified. The motivation was to reduce the time to develop a scheduler for an application as compared to the AppLeS scheduler. The GrADS scheduler however is centralized and makes scheduling decisions for the entire set of resources.

The GrADS project is also developing a meta-scheduler [5] that tries to mediate the requirements of multiple applications by evaluating the performance gain to the entire system if long running jobs are stopped to allow for jobs with shorter lifetimes to execute. The main component of this system is a contract negotiator that acts as a queue manager and mediates access to resources by multiple applications. The meta-scheduler has a database that maintains the state of running applications and a permission service that checks if an application submitted will be allowed to execute. The zone-based scheduler described in this thesis presumes that the load at the resources is representative of the other processes executing on those systems and hence does not take into account scheduling of multiple applications to balance their requirements. Also, meta-schedulers like the GrADS meta-scheduler [5] try to balance requirements of multiple applications by stopping and restarting jobs. In the zone based scheduler, such operations of stopping and restarting applications should not be allowed as the applications could be running on a remote zone and such operations cannot be performed.

The Legion system has a modular scheduler framework [7]. The scheduler in this system is basically an interface provided by Legion that needs to be implemented. Legion implements certain basic 'default' schedulers that are naïve; but the system itself is extensible in that an administrator or user can implement more complex scheduling mechanisms, via the provided interface.

The Condor project [36] is a system that harnesses idle computers. An application's requirements are specified by means of a ClassAd. ClassAds is a language that was designed to allow applications to find the resources that they require. There is a matchmaker component that performs scheduling in terms of matching resources to application requests. The Condor system seeks to maximize throughput of the entire system. The scheduler developed in this project seeks to improve the performance of a single application.

The Nimrod-G resource broker [37] uses an economic model for resource management and scheduling. Users can specify different requirements such as deadline, budget and optimization strategy. Nimrod-G dynamically trades with resource owner agents to select appropriate resources. The zone-based scheduler developed in this project does not take monetary issues into consideration, but such a facility could be provided by means of formation of virtual organizations that could impose economic constraints on use of resources.

In [52], Weissman et al. describe a federated model for scheduling in wide area systems. This system offers multiple levels of scheduling. Each level has a scheduling manager, which can interact with local schedulers present at the sites. However this method of scheduling takes advantage of information about the application provided by the user to a large extent. Also, contact information of site managers needs to be manually configured. In the ZBS framework, the information about the application is obtained from prior runs and schedulers controlling an organization's resources are automatically discovered.

One of the aims of this research is to explore scheduling across multiple administrative domains. In our project we use a scheduling mechanism that can be generically used for all applications, but which does take into account some application specific characteristics. The important point is that this application-specific information is relatively easy to acquire, that it can be improved with multiple runs, and that it reflects the realities of the grid-computing context. Thus, new applications can quickly be run on the grid efficiently. While it is generally accepted that application specific schedulers are likely to outperform a generic scheduler, we use our framework as a starting point from which distributed scheduling algorithms could be devised. The aim of this project is not to compete with application specific schedulers, but rather to provide acceptable schedules in the presence of administrative domains. Application specific algorithms could be used in future development of the scheduler as it involves just a new component to be introduced into the back end of the scheduler.

2.4 Scheduling Techniques and Methods

Scheduling algorithms used in grid applications range from the very simple to the very complex in terms of work performed to identify a mapping of tasks to resources. Since optimal scheduling of processes to resources is an NP hard problem except for trivial cases (and scheduling in the grid computing environment is not a trivial problem), almost all of these algorithms involve heuristics. Exhaustive search of the search space is possible only in cases where the number of resources and the number of processes to be mapped are small. This section provides a sample of typical algorithms used in scheduling processes. The zone-based scheduler developed in this research uses a few of these algorithms.

The literature available on the general topic of scheduling is enormous. Furthermore, there are many subtly different formulations of the scheduling problem. The most common formulation is that of a dependency graph. There are abundant scheduling strategies for scheduling dependency graphs [34, 35]. These papers take a set of nodes representing tasks and a set of directed edges between them representing precedence relationships between the various tasks. This dependency graph is then scheduled on processors in such a way as to minimize the critical path of the entire set of tasks. One of the main problems faced in this formulation of the problem is in finding the dependency graph of the program. Compiler driven methods by which this might be made possible without actually running the program have been considered in the POEMS project [33]. Another problem with this method of scheduling is that the task graph is enormous since every task dependency (where a task is a region of computation between two communication operations) needs to be taken into account. Thus, some form of condensing of the dependency graph is required.

Another formulation of the scheduling problem is that of representing the tasks in the form of an interaction graph. In this type of graph, the nodes represent processes and the edges represent intensity of interaction between those processes. For the rest of this thesis, this will be the problem formulation that will be used.

One way of performing the mapping of tasks to processors is to use a round robin strategy. For example, MPICH [43] uses a round-robin strategy to assign processes to processors. A 'machine file' is specified when the program is run. The processes are assigned to the processors listed in this file in a round robin fashion. MPICH-G2, the Globus enabled version of MPICH, uses this file to generate a mapping in the resource specification language (RSL) [19] format. In MPICH-G2 the user can also directly control the assignment of processes to processors by means of directly specifying the RSL file. These approaches to mapping processes to processors require the user to know the structure of the application and the resources in order to obtain a good mapping of tasks to resources. There are also other schedulers that choose random processors to place tasks on. Random strategies and variants of random strategies were used as a simplistic model in the Legion system [7] to compare their more advanced system.

The methods described in the previous paragraph do not take into account the differences in compute requirements at each task and the communication requirements between tasks. Better mappings are possible if this structure of the program is taken into account. The rest of this section describe efforts to include these factors into scheduling algorithms.

Clustering algorithms. Most scheduling algorithms described in this section have a preprocessing step in which they cluster the application interaction graph. An interaction graph is an abstraction of the application in which the nodes represent the processes (in our case MPI processes) and the edges represent communication operations between MPI processes. When clustering algorithms are applied to this graph, groups of processes that are closely coupled are identified (close coupling is determined by edge weights). This facilitates effective mapping of these clusters of the application onto resources with differences in the network latencies and bandwidths. We first look at some of these clustering algorithms and then describe the scheduling algorithms that make use of them.

There have been many heuristic clustering algorithms to partition graphs [30, 29, 20, 21]. Most of these algorithms were originally designed to partition VLSI circuits. Based on the number of clusters produced, the algorithms can be classified as either two-way partitioning algorithms or multi-way partitioning algorithms. The clustering algorithm may result in clusters that are of equal size or unequal sizes. Clustering algorithms can also be classified as being constructive (start with n clusters and hierarchically merge the closest clusters) or destructive (start with one cluster and hierarchically decompose the clusters).

Kernighan and Lin [30] proposed a solution that forms the basis of most iterative clustering algorithms. Kernighan's algorithm generates a two-way partition using a set of moves that lead to a better cluster solution after each iteration. This produces two clusters of approximately equal size.

Other clustering methods use annealing, which is a greedy approach. Sanchis [29] proposed a technique of selecting random seeds, with the remaining nodes annealed to the seeds based on a 'closeness' function between the different nodes. This method results in multiple clusters each of which is within a small range of sizes.

The algorithms described in the previous paragraphs make assumptions about either the size of the cluster or the number of clusters. Yeh et al. [20, 21] developed a clustering algorithm that makes no such assumptions. The algorithm is based on a clustering metric derived from the random graph model. The algorithm works by repeatedly injecting *flows* (a flow is a transmission of some data between nodes) between random nodes into the graph and disconnecting edges that saturate as a result of this flow. At the point when the graph becomes disconnected, a partition of the graph is obtained. This method effectively finds both the correct position and size of the clusters in the graph.

Heuristic Scheduling Algorithms. Orduna et al. [12, 13] have worked on the problem of incorporating communication requirements into scheduling decisions. They proposed a solution primarily for parallel tasks that have a high communication requirement and in which the communication forms a system bottleneck. In their algorithm, the set of mappings obtained by means of heuristic search methods (Tabu search) is evaluated based on a quality function. The quality function used is a clustering coefficient that evaluates the ratio of inter-cluster to intra-cluster communication costs. Their method is to be used in conjunction with a scheduling algorithm that takes only computation into account. The idea is to switch algorithms based on whether the task is a compute intensive or a communication intensive task, i.e., their algorithm is to be used for communication intensive tasks.

Taura and Chien's [22] algorithm takes into consideration the effects of both computation and communication of tasks simultaneously. Hence the same algorithm works for both compute intensive and communication intensive tasks and is better than taking each of those factors into consideration in isolation. The algorithm is aimed at providing better throughput for systems of pipelined processes. The algorithm goes through a preprocessing stage in which tasks are clustered to identify a set of tightly coupled processes. The clustering algorithm developed by Yeh et al. is used for this purpose. A tree of clusters is obtained by recursively applying the clustering algorithm. The tasks are ordered by performing a depth first search (DFS) on this tree. This DFS makes closely related tasks closer in the ordering obtained by this search. These tasks are then heuristically mapped, one by one, onto the set of processors, taking into account current bottlenecks and anticipated load on the machines and networks that have not yet been assigned any tasks. In addition, an "improve" step in their algorithm improved the final mapping by means of identifying bottlenecks in the graph. We used this algorithm and adapted it for our purposes by making it hierarchical (see Section 4.4.2).

Both Orduna's and Taura's work assume that a single parallel program communicates frequently and forms a single cluster. Their work aims to schedule many of these parallel programs on a set of resources where there is little interaction *between* parallel programs that form a data processing pipeline. However, within a single parallel program itself there are often processes that are closely coupled in terms of communication. This type of parallel program (i.e., one with different levels of communication requirements between the processes) is the target domain for the scheduler developed in our project.

2.5 Application Performance Monitoring Tools

The zone-based scheduler developed in this project obtains information about an application based on the previous runs of that application. This information about the application is used in subsequent runs to improve scheduling decisions. In order to be able to obtain this runtime information, there is a need to have sensors added to the application. There are two aspects to obtaining runtime performance information. The first aspect is the infrastructure required to collect all the runtime information required and to provide it to the user or system. The second aspect is determining what runtime data needs to be collected at each process in the parallel program. Several tools available to monitor and report information about the applications are described as follows.

MPI's Profiling Interface. MPICH offers a profiling wrapper to all MPI function calls. The profiling wrapper is implemented by means of weak aliasing the MPI function calls so that the profiled version of the function call gets called before the actual MPI function performing the communication. In order to collect application specific data this wrapper code needs to be modified, and any additional work that needs to be performed when this MPI function call is invoked is put in the wrapper function. Each of the MPI processes writes its profiling information to a local file. All these local files from all MPI processes are coalesced at the end of program execution by means of an All_Reduce operation taking place on the profiled data. This All_Reduce operation collects all the profiled information to the process with rank 0 which in turn writes the information to a consolidated file.

PAPI. PAPI, or performance API [23], provides a means by which a program can obtain low-level information about execution details. The program can make calls to the PAPI functions to return information such as number of instructions executed, execution time, number of cache misses, etc. Parallel programs use the information gathered by this tool to collect statistics related to the program's performance. The programmer then uses these statistics to identify bottlenecks in the program and rectify them. PAPI is used in a different manner in the research described in this thesis. We use PAPI to obtain computation requirements of the application, which will be used in scheduling decisions (see Section 4.2.1).

SvPablo. SvPablo [24] is a tool that provides access to low-level information (similar to PAPI) as well as combining the information at the end of the program's execution. SvPablo uses the compiler to instrument the application automatically to compute performance metrics. This has the advantage that the programmer does not need to change the source code to incorporate function calls like in PAPI. SvPablo offers binding in multiple languages and has a standard format for profiling messages called SDDF (self describing data format).

AutoPilot. The AutoPilot system [15] is an adaptive steering system that allows adapting the execution of the current run of program rather than providing information after the program has finished execution. The autopilot system works by introducing distributed lightweight performance sensors that capture quantitative application and system performance data. Since the information is being gathered *and used* at run-time, the amount of information collected is not a complete trace, but some synthesized metric [14] that will be useful in steering the process.

In order for a scheduler to make effective scheduling decisions, a model of the application's performance is required. The application performance model needs to be obtained in some way – either provided by the user (e.g., in AppLeS the user provides the performance model) or extracted by the compiler (e.g., in the POEMS and the GrADS project the compiler automatically obtains this information) or obtained from previous runs of the application. The third method requires sensors at the various processes, collating infrastructure to coalesce the information for one application, and a storage facility to store this information so that it can be provided to the scheduler on the next run of the application. In our scheduler design, we use the third method of obtaining information about the application and hence we need to use application performance monitoring tools such as those described in this section.

2.6 Web Service Frameworks

Web service frameworks have become popular as a means to provide functionality that is easily extendible and in which it is easy to interface various software components. Such frameworks reduce the effort for the programmer to integrate various software and are based on vendor-neutral standards that are implementation, language and operating system independent. Such a framework is necessary to implement the zone-based scheduler described in this project due to the following reasons. Firstly, the zone-based scheduler is distributed and there is a need to transparently find and interface with other peer schedulers. Secondly, the zone-based scheduler was designed to interact with other grid services. Examples of such services are run time algorithm selection systems and rescheduling systems. There is a need for all these systems to interact transparently with each other, which is facilitated by the web service philosophy. The main web service technologies that are used in the implementation of the zone-based scheduler are described as follows.

UDDI. UDDI (Universal Discovery, Description and Integration protocol) [25] is used to access registries that store information about services. UDDI registries store information about web services available and registered with the UDDI directory. The zone schedulers from different zones (described in Section 3.1) register with this UDDI registry. Also the zone-based scheduler interacts with other grid middleware if they are offered as web services and are registered with the UDDI registry. UDDI stores 3 classes of information about a service.

1. **White pages**, which contain information about the organization to which the service belongs, and contact information of that organization.
2. **Yellow Pages,** which allow search for all web services that fall under a particular class of applications. Thus the zone-based scheduler could search the UDDI registry for all services that perform scheduling of grid applications.
3. **Green Pages,** which provide the technical specification and interfaces to the web service as a WSDL (Web Service Description Language) file. Using this file, software can verify if it can interface to another piece of software by means of the APIs it provides.

Web Server Technologies. Apache Tomcat [26] was used as an application container that provides the zone scheduler web service. Tomcat is a web server that can be contacted by means of the HTTP protocol on port 8080. The tomcat server not only provides access to web pages but also has active elements in the form of server side programs. Any service, like the scheduler service, that is to be provided is a program that is stored in tomcat.

Apache Axis toolkit [27] is a SOAP (Simple Object Access Protocol) [28] generation engine. Essentially the application (in this case, the zone-based scheduler) is written and then tools provided by the Axis toolkit are used to automatically generate wrappers for this code to allow for seamless integration with other software. The wrappers to the client and server codes are similar to stubs and skeletons generated for technologies like COM and CORBA. The main advantage of the wrappers generated here is that they are in XML format and hence make integration very easy. These wrappers handle the conversion of input and output parameters of the web service to a vendor-neutral format (i.e., XML). The toolkit then communicates all information pertaining to interaction of this web service with other software components by means of SOAP messages which are both operating system and language independent. Programs (e.g., Java2WSDL) are also available in the Axis toolkit to automatically extract WDSL information given a Java program.

GPDK. GPDK (Grid Portal Development Kit) [44] is a toolkit that provides a set of reusable components for accessing various grid services. It provides interfaces to allow remote program submission, file staging, and querying of information services. It facilitates easy access to grid services by a lightweight client in the form of a web browser. The browser contacts a web server, which has JSP pages and servlets to interface to grid services. The zone-based scheduler developed in this research could be integrated into portals like GPDK. In order to facilitate this integration, the thin scheduler client provided needs to be modified to a ‘HTML form’ and a wrapper to the scheduler code needs to be written in the form of JSP or servlets.

Chapter 3. Scheduler Design and Implementation

Zone based scheduling is a new method of scheduling based on organization domains. To clarify the presentation, a set of important terms is defined in Section 3.1. The grid has taken on different meanings to different people. Section 3.2 describes the type of problems being addressed here and the niche area for this kind of scheduling methodology. Finally, the design and implementation of the scheduling framework is described in Sections 3.3 – 3.5.

3.1 Terminology and Definitions

Zone: A zone comprises a set of machines and interconnection networks that is owned by or controlled by a single entity. Since a single entity controls all resources within a zone, the resource allocation policies and mechanisms for these resources can be made at a single point within this domain.

Scheduler: The term scheduler is used in this thesis to represent the framework or the overall architecture in which specific scheduling algorithms may be implemented. This is differentiated from instances of the actual scheduling algorithms themselves (e.g., described in Chapter 4). In particular, when referring to the scheduling framework developed in this research we use the term ***Zone Based Scheduler (ZBS)***.

Scheduling Algorithm: A scheduling algorithm is a particular algorithm that makes the scheduling decisions within or across zones. This might be some algorithm described in Chapter 4 to perform scheduling or some other mechanism or algorithm defined by the administrator of a particular resource.

Zone Scheduler: A zone scheduler is a server that users contact to make requests to schedule applications. Each organizational domain (*zone*) has a zone scheduler. This scheduler reflects the resource allocation policies and scheduling algorithms that are suited to that organization's requirements. The zone scheduler offers a web service interface to other zones. A zone scheduler could be unique to a particular zone's requirements or it could simply be a wrapper to existing schedulers like PBS/MAUI [31, 32].

Home Scheduler: A home scheduler is the scheduler on the zone where a user submits an application. The Uniform Resource Identifier (URI) of that client's home scheduler is the default location a client contacts to make a scheduling request. Thus, home scheduler is the term used to refer to the zone scheduler corresponding to a particular user's organization.

SPMD Program or Application: An SPMD (single program, multiple data) program is a parallel program consisting of interacting processes. There is a single code base for all the processes, but the control path that each process takes may be different. A typical motivating application used in this research is an SPMD program that solves a system of linear equations using ScaLAPACK, which uses MPI internally to perform message passing.

Task or Process: This corresponds to one single control flow in a parallel program that may have many control flows. More specifically, it corresponds to an MPI process with a particular rank or identity. The term task is used synonymously with process.

3.2 Scope and Relationship to Current Systems

The GGF document "Ten Actions when SuperScheduling" [1] describes the various stages involved in scheduling an application. The ten steps identified in that document are authorization filtering, application requirement definition, minimal requirement filtering, information gathering, system selection, advance reservation, job submission, task preparation, task monitoring, and job completion. The system described in this thesis performs the equivalent of the first five of these steps, though not necessarily in that order (due to the nature of the distributed scheduler). The scope of the ZBS system starts at the point when a request to schedule an application arrives and ends when the scheduler provides a mapping of processes to processors. The output of the scheduler is an RSL file [19]. The RSL file is given to Globus, which handles job submission, job monitoring and input/output file staging.

The target applications for the zone-based scheduler are SPMD programs that are run multiple times. Examples of such applications are parameter sweep experiments where the application needs to be run for ranges of multiple parameters. This allows the performance information from prior runs of the application to be used in making scheduling decisions for the next run of the same application. An implicit assumption made here, which is typical of the target applications, is that the communication and computation requirements of the application do not vary significantly from run to run. Also, the zone-based scheduler has been designed with a *multilevel* or *clustered* communication structure in mind. This means that the processes of an application can be split into subsets of processes. Each subset communicates intensively within its cluster but there is relatively little interaction between clusters.

The zone-based scheduler developed in this project schedules applications for execution immediately. While the proposed architecture allows for schedulers to assign processes to resources at a later time, the current implementation does not handle this case. The problem of co-allocating resources at a future time is not addressed in this research. Thus the current implementation works for timeshared machines and for machines with batch schedulers that can run jobs immediately, i.e., the batch scheduler has free resources at the moment the scheduling is to be performed.

Reservation of resources for the duration of the application's execution time helps to guarantee the performance of the application. While this is a desirable feature, not many existing systems guarantee quality of service based on resource reservation. We use NWS to forecast the processor and network characteristics for the duration of the run. This forecast is used in all scheduling decisions. If the performance of a resource is predictable (e.g., a workstation classroom known to be busy from 9 am to 5 pm only) such a strategy works well. However on resources that have unpredictable fluctuations in load such a forecast is not very accurate. The ZBS approach works well for systems with predictable workload characteristics, or for systems with unpredictable workloads having reservation.

Since the ZBS framework is not designed for systems whose applications are scheduled to run in the future, we do not address the issue of locking of resources to prevent race conditions. If applications need to be scheduled on batch systems to be executed in the future, there is a need for advance reservation on those resources. For the simpler case where all applications are scheduled to execute immediately, the race condition is not as pronounced.

Our scheduling methodology is adaptive in the sense that the scheduler takes into account load on the machines and networks at the time of scheduling. However, once a schedule is obtained, the schedule is not modified to reflect changing grid conditions—the scheduler is not adaptive in this respect.

The zone-based scheduler has been designed to be as transparent to the application and the user as possible. Currently, a command line script (mpirun) is executed to run MPI based programs. A client GUI has been implemented (in Java Swing) to allow the user to specify the files containing the input program and the performance information from prior runs. The client tool then performs everything from contacting the user's home scheduler to executing the script that runs the MPI based program

3.3 Design Constraints and Criteria

Each organization has a scheduler called the zone scheduler. This scheduler is responsible for making the scheduling decision for all resources in that zone. This kind of a design is important to allow each zone to maintain control of all resources in its own domain. The resource allocation policies and scheduling mechanisms are incorporated in the scheduler for that zone.

The ZBS architecture was designed to solve a set of problems that are related to the pragmatic aspects of the grid. Many of these issues arise due to the organizational nature of the grid and the ways they are addressed are described as follows.

Scalability of Information Gathering. In order to be able to make scheduling decisions, some infrastructure is required to obtain and maintain up-to-date information about the state of the resources. Grid monitoring systems like MDS allow for obtaining this information. MDS uses a push model to give information about load at various machines and networks periodically. To improve the scalability of the solution, MDS has used mechanisms like caching of entries. In the zone-based approach, a pull method of obtaining resource information is used. A pull model is acceptable due to the web service discovery of peer zones, due to the nature of the application and due to the type of users. The applications being targeted here are expected to take large amounts of time to execute with requests to run an application coming at relatively infrequent intervals. This implies that a pull model of obtaining information about the state of the resources is more suited to this domain.

Scalability of Scheduling Algorithms. Scheduling algorithms take as input the set of tasks of an application and map them to a set of machines. For non-trivial scheduling algorithms (trivial scheduling strategies include random and round-robin strategies), the complexity of these algorithms is proportional to both the number of resources and number of tasks. The ZBS framework uses a divide and conquer approach by splitting the set of tasks based on communication intensity and splitting the set of resources based on organizational boundaries. Thus the number of tasks and the number of resources for each scheduling decision is small which reduces the overall running time of the scheduling algorithm and increases scalability. Also, scheduling decisions can now be made in parallel at many sites. Both these factors reduce scheduling time and hence reduce the possibility of applications interfering with one another in a distributed environment due to race conditions.

User Accounts. Grid users are required to have user accounts on machines where their program runs. Schedulers filter out the machines on which the user does not have an account from the resource set during the scheduling process. Thus the scheduler needs some mechanism of answering the question—*Does the user whose application is being scheduled have an account on machine X?* MDS [39] does not store user account information in its directory. MDS does not store this information as it represents a security hole if such information can be queried. The main reason for this problem is due to the nature of MDS, i.e., it can be queried by anyone without restrictions. However, there have been recent improvements to MDS to allow for authenticated querying of MDS information, which might allow storing account information for the users.

Currently existing schedulers, like the GrADS scheduler, circumvent this problem by having the user provide a machine list to the scheduler. These machines are the only machines on which the user has an account. Thus, the user performs filtering explicitly. However, providing the machine list places additional burden on the user and does not facilitate the easy formation of dynamic virtual organizations.

The need for user account information at the resources is obviated in the zone-based scheduling framework. Since each zone makes scheduling decisions for the domain it controls, information about which user has an account on which machines is known to that zone's scheduler. User account information can be maintained at the zone scheduler since the zone scheduler is trusted by the organization. The user account information need no longer be queried by anyone, but only by the scheduler in the organization.

Control. One of the major aspects of scheduling is the location where the policy decisions will be made. Traditionally there have been two locations of control. The first location is with the user. At this location of control, when a user needs to schedule an application he invokes a scheduling agent that either knows about the state of resources on which the user can run the application or goes about obtaining this information.

Scheduling agents have been very common in the past and they permit very fine-grained task allocation since they have a detailed knowledge of the application's structure [3, 18]. The other location of control is at the resource itself. This is typical of batch scheduling systems that try to obtain best performance for many applications, thereby increasing throughput [5]. The ZBS framework proposed in this research offers an intermediary between the two typical locations of control. In the zone-based framework, each organization's resources have a scheduler that acts as a policy enforcer for scheduling resources in that domain. In addition, the home scheduler has the user agent incorporated in it to manage the application's overall requirements and to coordinate the schedules arriving from different zones.

Transparency. The intrusiveness of a scheduler on the user is one metric to determine the usefulness of the scheduler. Many schedulers require a performance model of the application [3, 18, 4, 6]. This performance model contains problem parameters that directly affect the running time of the application. However, the user needs to perform additional work to specify this performance model. Other schedulers use compile-time techniques to extract this information, which reduces the burden on the programmer. The scheduler designed in this project is transparent to the user and the execution model of the application is obtained from runtime techniques described in Section 4.2.1.

Platform and Software Independence. Language and operating system independence provided by the web service framework is an advantage of using the ZBS framework; in this way each organization can follow its own policies, algorithms, language and operating system. The only component necessary for two organizations to combine their resources is for an agreement to be reached to collaborate. In essence, the formation of virtual organizations must ideally involve only administrative issues. Using the web service framework, such an ideal of collaborating services between organizations can be realized.

3.4 Stages in Application Scheduling

This section presents an overview of the sequence of steps by which a process gets scheduled on resources using the zone-based design. We present the stages an application proceeds through to get from the user submitting a request to getting the program run and recording performance information. Further details regarding these steps are given in subsequent sections.

The various phases to executing an application are the registration phase, the scheduling phase and the job execution phase. These phases are described below. The registration phase takes place once at startup of the scheduler and periodically thereafter with a low frequency as the number of zones is not expected to change frequently. The scheduling phase and the job execution phase happens every time a user requests an application to be scheduled.

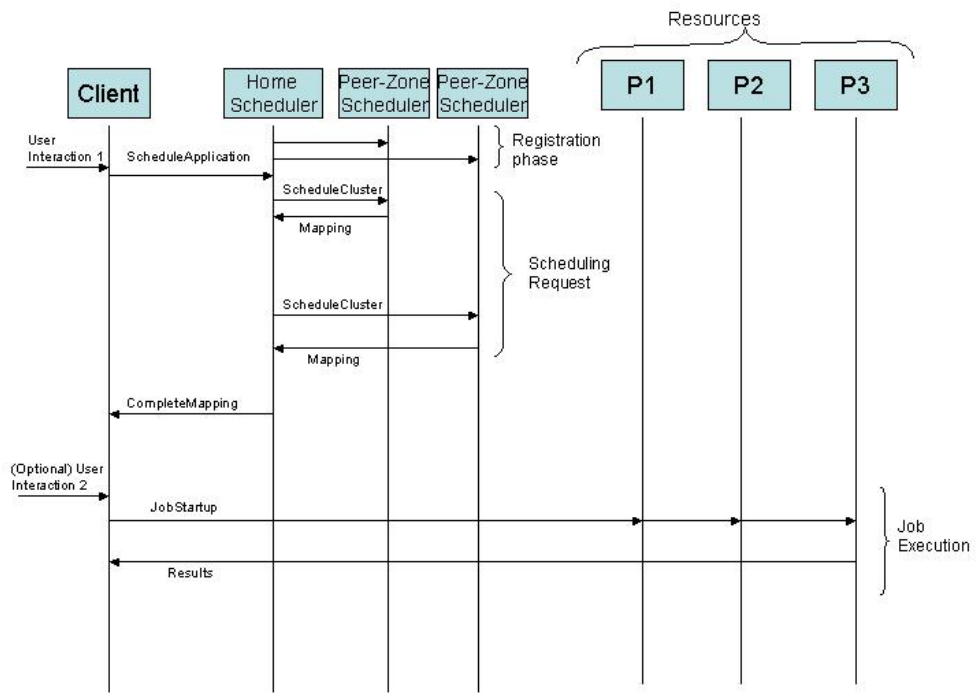


Figure 3.1. Timeline diagram of the scheduling process

Phase 1: Registration

- *Search for peer schedulers*

When the scheduler in a particular organization comes online, it contacts a global registry to determine if there are any similar peer schedulers. The aim of this step is to facilitate the discovery of organizations that are willing to contribute resources to the grid. From the global registry, a list of other zones or organizations is identified and technical information as to how to interface with them is obtained.

- *Register with peer schedulers*

Once peer schedulers have been identified, the next step is for the home scheduler to contact and register with them. The remote scheduler might decide to accept or reject the registration. If the remote scheduler accepts, then the remote scheduler in effect agrees to schedule processes of users belonging to that organization. This phase is necessary to allow for a controlled formation of virtual organizations. Performing this step allows each organization to explicitly specify which other organizations can schedule processes on its resources. The registration can be unidirectional or bi-directional. In the unidirectional case organization A can permit organization B’s processes to be scheduled on its resources whereas organization B need not allow organization A’s processes to run on its machines.

Phase 2: Scheduling

- *User requests to schedule an application*

In this step, the user has a program to be run. The user uses the client GUI tool to provide performance information obtained from previous runs (of the same program) to the home scheduler. This happens by invoking the *ScheduleApplication* call (shown in Figure 3.1) from the client to the home scheduler. The details of obtaining performance information are described in Section 4.2.1. After these inputs are obtained, the client contacts the home scheduler web service interface and transfers the performance information.

- *Obtain resource structure for the user*

When the user sends the information about the application to his home scheduler he also sends his proxy certificate. Essentially the home scheduler now can authenticate this user to all other peer schedulers using the single sign on facility. This is the step at which authorization filtering occurs. Schedulers can now reduce the set of resources about which information needs to be provided to only those resources on which the user has an account. Thus, processor and network load information returned to the home scheduler are only about those resources on which this particular user has an account.

- *Cluster application based on prior runs*

Clustering aims to identify tightly connected components of a graph. The aim of clustering in the context of scheduling is to be able to identify subsets (or "clusters") of processes that communicate a lot among themselves. The clustering operation is based on the application's communication patterns from previous runs, and results in a certain number of clusters that is not fixed ahead of time. Thus, the tasks of an application are split into clusters, with processes in each cluster interacting heavily and relatively infrequently with processes in other clusters.

- *Schedule using information about process clusters*

Once the clusters have been identified, the next step involves scheduling each of these groups of tasks. If necessary, separate clusters will be scheduled on separate zones. This happens by the home scheduler sending out requests to its peer schedulers to schedule one or more clusters (*ScheduleCluster* in Figure 3.1). Each zone is responsible for making its own scheduling decision. The peer schedulers return a mapping to the home scheduler, which consolidates the results into one large mapping for the entire application. Thus, there are two components to making scheduling decisions, one is assigning process clusters to zones and the second is assigning processes within a cluster to machines in a particular zone. The precise mechanisms of scheduling are described in detail and evaluated in Chapter 4.

Phase 3: Job Execution

- *Run the program on the selected set of machines*

Once the mapping is complete, it is returned to the user. The application is then run on the set of resources identified in the mapping by means of job startup mechanisms provided in Globus. This is achieved by means of an extra step at the client side to submit a job using mpirun (user interaction 2 in Figure 3.1). This could also be automatically performed at the point when the user makes a request to schedule the application (user interaction 1 in Figure 3.1).

- *Obtain and use performance information*

The application is linked with sensors that provide information about the structure of the application. These sensors are present in MPI, which we have augmented to provide additional information for our purposes. Thus the user does not have to modify or insert any additional statements in the code to obtain this performance information. This allows the scheduler to be as transparent as possible to the user. A more detailed explanation of how the performance information is obtained is given in Chapter 4. This performance information is stored and the user provides it to the scheduler when the same program is executed the next time.

3.5 Web Service Architecture and Implementation

In this section the architecture and the software components used to implement the ZBS system are described. The architecture is a hybrid of client-server and peer-to-peer architectures. The user interacts in a client-server manner with a home scheduler. The schedulers in different organizations interact with each other in a peer-to-peer manner. Integrating both these paradigms is made possible by means of the web service infrastructure.

There are two software components that need to be implemented to use the zone-based framework. The two components are the client and the server. The implementation of these components and design decisions related to implementation are explained in the next few sections.

3.5.1 Client Implementation

The implementation of the zone-based scheduler has striven to be thin-client based with most of the complex logic and software present on the server in a zone. This design choice was made to keep the client side as platform independent as possible. A prototype was implemented in Java. However the client can be implemented in any language since the scheduler in a zone is a web service. The client contacts this server by invoking one of the scheduler's functions.

When the client side contacts the home scheduler, the client provides the executable name and the performance model of the application. In the prototype implementation, the performance model of the application is extracted from a simple ASCII log file obtained from the profiling interface of MPICH. The profiling interface in MPICH has been augmented to record collective communication operations as point-to-point operations (collective operations take place as point-to-point operations internally in MPI). This allows the scheduler to determine which processes interact frequently. If collective communication operations were not recorded in this manner, it would be difficult to characterize multiple collective operations properly.

```
L.No  MsgType              TimeStamp       Src         Dest      Bytes
1     -101     0  0  4  0  1093114    S     0    D      4     SZ  200
2     -101     0  0  1  0  1093150    S     0    D      1     SZ  200
3     1        0  0  0  0  1093220    PAPI  0    52723
4     -101     0  0  4  0  1093250    S     0    D      4     SZ  200
5     -101     0  0  1  0  1093289    S     0    D      1     SZ  200
6     -102     0  0  0  0  1722303    R     1    D      0         720
7     -102     0  0  0  0  1722344    R     2    D      0         720
```

Figure 3.2. Sample augmented ASCII log file

Figure 3.2 shows a sample log file that has been augmented with additional information for the purposes of the zone-based scheduler. The second column represents the type of the event that took place in the execution of the program (-101 represents a send event, -102 represents a receive event, 1 is any general event that can be recorded by the profiling interface—in Figure 3.2 it is used to represent the size of a compute block recorded by PAPI). The other fields of interest in the log file for a send or a receive event are the source, destination and size of the message. The last entry in line 3 represents the execution time that the process took between successive communication operations.

The log file can be extremely large for long running programs. Hence when the client contacts the home scheduler, the client needs to provide only the model of the application (explained in Section 4.2.1); it is not necessary to provide the entire file. Thus the processing of the ASCII log file takes place on the client side and the only information that is transmitted to the server side are the computation requirements at the various processes and the communication requirements between processes.

In more advanced versions of the scheduler, more sophisticated performance models of the application could be provided. Different scheduling algorithms could be used at the server side for any given representation of the performance model of the application. The intricacies of the scheduling algorithm used at the server side are transparent to the client. Hence, the user does not need to make any changes to either his code or his mode of interaction with the scheduler when there is a change to the scheduling mechanism at the server.

3.5.2 Server Implementation

Scheduler implementations run within a web server container. The web server runs on the HTTP port and hence can be operated through firewalls. This implies that a user of the scheduler can be located outside the organization and can still make requests to schedule an application. The web server used for this project is the Tomcat web server.

The server is designed to allow organizations to easily discover other organizations that are willing to collaborate by sharing their resources. When the server starts up, it contacts a UDDI registry to search for other organizations, which have a similar peer scheduler controlling the resources in another administrative domain.

An organization that desires to collaborate on the grid implements a scheduling algorithm corresponding to its own scheduling policies. Web services are based on XML standards and hence allow for interoperability between software written in different languages. Thus the scheduling algorithm can be implemented in any language and operating system.

The scheduler implementation is compiled and the executable class files obtained are archived into a java archive (jar file). The java archive is deployed using the *AdminClient* tool available in the Axis toolkit [27]. This deployed web service can be accessed either by a client as developed in this project or by means of an HTML form in a web browser.

The implementation of the scheduler can be differentiated from the interfaces it offers. The implementation of the scheduler can vary significantly even though a uniform interface is offered to other entities. The only restriction on this transparency is that the application representation must be fixed ahead of time. We describe how we represent the application in the scheduling algorithms implemented in this project in Section 4.1. The interfaces offered by the server are as follows. There are three methods that can be accessed by other software.

- Register

  This method is invoked when virtual organizations are looking for collaborations and resources on which to execute their parallel programs. This allows for secure sharing of resources in that the schedulers can be mutually authenticated. If the collaboration is taking place for the first time, then temporary accounts could be made on the resources available. This is possible by means of a dynamic account creation mechanism and is a current area of research in GGF [40].

- ScheduleApplication

  The client invokes this method to provide the application and the performance model of the application to the home scheduler. This method starts the process of clustering of the tasks of the application and the search for resource load information on all the resources on which that user has an account. It then schedules on the set of resources by contacting the peer schedulers.

- ScheduleCluster

  The peer scheduler invokes this method to schedule a group of processes on the set of resources owned by the scheduler at which the method is invoked. This method uses the performance model of the group of processes and the scheduling algorithms and policies of the organization it represents to perform the mapping of tasks to resources.

Chapter 4. Scheduling Algorithms

While the framework described in Chapter 3 is extensible and many different scheduling mechanisms can be incorporated, we have evaluated the framework with a small set of scheduling heuristics. Section 4.1 introduces a formulation of the scheduling problem that is used in scheduling strategies evaluated in this project. Section 4.2 provides an explanation of how the application model and the resource model are obtained. Sections 4.3 and 4.4 describe two scheduling heuristics. Finally, Section 4.5 makes a qualitative comparison of the different scheduling mechanisms.

4.1 Problem Formulation

The formulation of the scheduling problem used in the scheduling algorithms implemented in this research is that of the *resource-task* graph. This is the same formulation used by Taura et al. [22]. In this formulation of the scheduling problem there are two inputs to the scheduler – the resource graph and the task graph. The scheduler is responsible for mapping the nodes of the task graph to the nodes of the resource graph.

Resource Graph: A resource graph is a graph weighted on the nodes as well as the edges. A node corresponds to a single machine or processor and an edge corresponds to the interconnection link between two machines. The weight on the node corresponds to the CPU speed available to a process that is started on that machine and the weight on the edge is extracted from network parameters between those two machines (the network parameters used and how they are extracted are described in Section 4.2.2).

Task Graph: A task graph is a graph weighted on the nodes as well as the edges. A node corresponds to a single process and an edge between two nodes represents the interaction between the two processes. The weight on the node corresponds to how computationally intensive that process is and the weight on the edge corresponds to the amount of communication between those two processes. This graph could also be called the interaction graph and represents a model of the application.

An important point to note about both the task and the resource graph is that the weights on the nodes and edges can be obtained in many different ways. Also, only the ***relative weights*** among the nodes or among the edges are of importance to the algorithms evaluated in this chapter. As an instance, we could use CPU speed (metric 1) or RAM size (metric 2) as the weight of a node in the resource graph as they are both representative metrics of how quickly the program executes. If both of these metrics yield the same weights for each node in the resource graph, the scheduling algorithm will perform the mapping in exactly the same manner if either metric is used. Thus, the precise metric used is not as important as the relative weights between the nodes. In future versions of the scheduler, better and more accurate ways of obtaining the weights could be designed (like some weighted average between CPU speed and RAM size).

4.2 Obtaining Scheduler Inputs

The task graph is obtained from the application and the resource graph is obtained from the sensors present on the machines. These represent models of the application and resources respectively. First, we look at the lifecycle of an application run on grid environments and then we look at obtaining the application and resource models.

The lifecycle of an application running on grid environments is as shown in Figure 4.1. The application model is given as input to the scheduler. This corresponds to the computation-communication structure of the application. How this information is extracted from the application is described in Section 4.2.1. Using the application's information and the information about the current load at the various machines and networks, i.e., the resource model (described in Section 4.2.2), the scheduling algorithm makes an assignment of tasks to machines. This mapping is provided to job startup mechanisms present in execution environments like Globus. The job startup mechanisms present in Globus take care of co-allocation of the various resources. Once the application's execution is completed, the performance of how well the application ran on the given set of resources is obtained. This performance information could be used by the zone-based scheduler the next time the same application is to be scheduled.

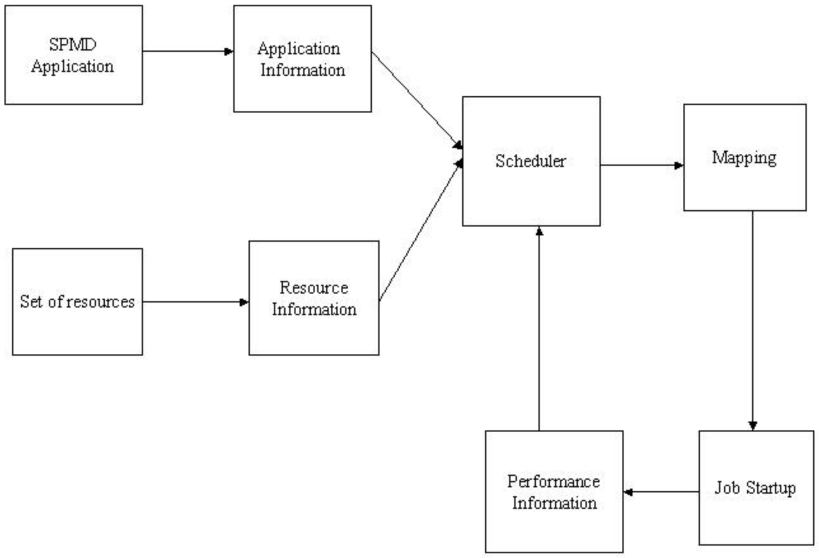


Figure 4.1. Lifecycle of an application.

4.2.1 Application Model

In order to schedule an application the scheduling algorithm needs a structural model of the execution of the application. This structural model aids the scheduler in the placement of tasks. There are three ways to obtain this information, which are described as follows.

1. **User Supplied**

   The user might supply the information about the structure of the application to the scheduler. This is the method used in AppLeS and GrADS schedulers. This information is in the form of the user specifying what problem parameters affect the application and how the execution of the application changes as a result of these parameters. Thus the scheduler would be able to use these analytic models of the application to predict compute intensive and communication intensive tasks and accordingly place the tasks. The disadvantage with this method is that it places additional burden on the user.

2. **Compiler Extracted**

   The POEMS [33] project and the GrADS project have a compiler to extract the structural model of an application. The POEMS representation of the application is useful for both simulation and analytic models. The difficulty with this approach is that runtime conditions cannot be predicted. Thus, dynamically determined loop control variables and variables controlling conditional statements cannot be obtained in this approach. Also, the amount and destination of communication could be determined by runtime conditions, and hence cannot be predicted.

3. **Runtime System Extracted**

   Another method by which the structure of an application can be obtained is by means of executing the application and obtaining information from the execution. This is the method we use in the zone-based scheduler and is explained in detail in the next few paragraphs.

   There are two basic criteria for using this method for extracting the structural model of the application. Firstly, this information can only be used if future executions of the application have a similar structure to prior executions of the application. This is typically the case for parameter sweep experiments for example. Secondly, the information gathered must be carefully chosen so that idiosyncrasies of the current environment do not skew the structural model of the application. Basically this involves removing or reducing noise from the information gathered at runtime.

   For the purposes of the zone-based scheduler we used basic metrics that are not likely to be influenced by the environment. However, these metrics might be simplistic and there might be need to obtain better metrics in the future. Based on the problem formulation described in Section 4.1, there are two classes of information to be obtained about the application's structure – the weight of an edge and the weight of a node. The weight of a node is defined as a task's compute requirement. We used the average time of computation between two successive communication operations as an estimate of the compute requirements for a given node. The weight of an edge connecting two tasks should reflect how tightly coupled those two tasks are, and hence how important it is to put those two processes close to each other. We used the number of bytes communicated between the pair of processes as a measure of this coupling. This is preferred over using methods like *MPI communicators* to determine the tight coupling between processes. An MPI communicator is a set of processes over which collective communication operations take place. Thus if processes belong to the same communicator they could be construed as having a tighter coupling. However, since the amount of communication within different communicators might vary and there might be processes that are a part of multiple communicators, such a method is ineffective.

In order to be able to obtain information about the application at runtime, there must be sensors inserted into the application, which record information about the application's execution. As explained in the last paragraph, two pieces of information are recorded by the sensors – the average time spent in computation between communication steps and the number of bytes communicated between a pair of processes. How each of these pieces of information is obtained is explained in the next few paragraphs.

Computation. In order to measure this metric, it was necessary to measure only the time spent doing useful work in the current process, e.g., ignoring system or wait time. Thus using MPI's profiling interface was not possible since it uses wall clock time. If there are multiple processes on a particular multitasking machine (a likely scenario in grid environments), then wall clock time of a particular process is not an indication of how compute intensive that process is. In order to get the compute time of only the application under consideration, Performance API (PAPI) calls were inserted into MPI functions. Currently, these calls write the difference in time between the last MPI call and the current MPI call into the log file. If these differences are large we assume a more compute intensive task. Since PAPI only records the execution time for the current process, other processes executing on the same machine will not affect the metric.

PAPI could be used to obtain more sophisticated factors about the structure of the application. PAPI allows for recording fine-grained information about the application like number of instructions executed, number of cache misses etc. Such information could be used in a future version of the scheduler to make more accurate predictions about an application's structure. This flexibility offered by PAPI led to its use in this project rather than using functions that return system time.

Communication. MPI communication operations are of two types – collective communication and point-to-point communication. Collective communications in MPICH-G2 are implemented as a set of point-to-point operations. Thus all communication can be broken up into process-to-process communication. The augmented MPI's profiling interface provided this information along with the rank of the source and destination of the communication and the number of bytes transmitted. This served as a basis for determining the edge weights on the task graph.

4.2.2 Resource Model

The second input that a scheduler needs to perform its tasks is a model of the structure of the resources. We used a simple model of the resources; but, as in the case of the application model, it can be extended to take more complex issues into consideration.

In the resource graph, the weight on a node corresponds to CPU speed and the weight on an edge corresponds to the reciprocal of the end-to-end delay connecting the processors (end-to-end delay is determined by latency and bandwidth of the links connecting the processors). Note that in the experiments described in Chapter 5, both the node weights and edge weights in the resource graph are normalized with respect to the largest node or edge weight respectively, and the edge weight is scaled up by a constant factor. The reason this is acceptable is because only the relative weights are important and both normalization and scaling does not affect the relative weights.

The resource model is obtained by using MDS and NWS. NWS makes forecasts about load and hence can be used to predict load at processors and networks for the duration of a program's execution.

4.3 Clustering Heuristics

A summary of various clustering heuristics is given in Section 2.4. In this section we explain the algorithm used in this project to cluster tasks. This clustering algorithm was originally designed by Yeh et al. [20, 21], and was later modified by Taura et al. [22], for scheduling purposes (described in the next section).

The input to the clustering algorithm is the task graph. Recall that the task graph has weights on the nodes as well as the edges. Edge weights in the task graph correspond to the amount of communication between processes. These edge weights are of importance to the clustering algorithm. The clustering algorithm, called shortest path clustering (Figure 4.2), has two steps—Saturate Network and Select Cut.

Saturate Network. In the task graph, the *capacity* of an edge is the weight on that edge. Thus, the amount of communication between processes in the task graph represents the capacity of the edges. The algorithm repeatedly selects two nodes randomly and injects a small *flow* of data along the edges in shortest path between those two nodes. Thus each edge during the algorithm's execution has a capacity and a flow. The flow between the two selected nodes increases (as more and more small flows of data are injected between randomly selected nodes) until the flow along at least one edge exceeds its capacity (the edge is said to saturate).

Select Cut. The saturated edges represent the points in the graph at which cuts can be introduced to partition the graph. The cuts are made in the graph by removing edges that belong to the *set of saturated edges* using a cluster ratio metric. The cluster ratio metric (defined in [20]) is used as a measure by which inter-cluster communication is minimized. The minimum value for the cluster ratio metric implies that the best clusters are obtained. This makes cuts through the components of the graph that do not have a lot of communication.

```
// Circuit N; Flow increment is F; Distance Coefficient A
Algorithm ShortestPathClustering(N, F, A)
{
   do
   {
         randomly pick two nodes s and t.
         Find shortest path between s and t.
         Inject a flow between s and t.
         Remove edges where the flow exceeds its capacity.
   }while(clustering is not done)

}
```

Figure 4.2. Shortest Path Clustering

The clustering algorithm developed by Yeh et al. has a lower time complexity as compared to other clustering algorithms [20]. Also, this algorithm does not have any restrictions on the size or number of clusters. This is a very important requirement of an algorithm to be useful in clustering a task graph, since otherwise it might result in unnatural partitioning of the graph.

Given the task graph defining the application model, the home scheduler uses the algorithm just described to identify clusters. Once these clusters of tasks have been identified, the home scheduler assigns them to various zones.

4.4 Scheduling Heuristics

A summary of various scheduling heuristics is given in Section 2.4. In this section two scheduling heuristics are described in detail. The first heuristic is extracted from the work of Taura et al. and is described in Section 4.4.1. Section 4.4.2 describes how the algorithm designed by Taura is used in the context of the zone-based scheduling framework, and suggests modifications that help with some of the limitations of the original algorithm.

4.4.1 Taura's Algorithm

The algorithm developed by Taura et al. has three steps.

Cluster the task graph. Taura et al. uses the clustering algorithm developed by Yeh et al. One modification done to the original clustering algorithm developed by Yeh et al. is that the clustering is done recursively. The idea is to split the task graph into coarse clusters and then split those clusters into sub-clusters till each cluster is a singleton element. Then a depth first traversal of this tree of clusters (and recording the leaves of the tree) obtained from recursive clustering is performed to obtain an ordering of tasks. When this traversal is performed tasks that are closely coupled end up closer to each other in the ordering than tasks that are not closely coupled. This ordering of tasks is important due to the way the next step (the Map Tasks step) of the algorithm works.

Map Tasks. This step takes the processors in the resource graph and in sequence takes one processor and maps a set of tasks to it before moving on to the next processor. Since closely related tasks are closer in the ordering obtained from the traversal, it is very likely that processes that communicate frequently will be mapped to the same processor. The metric used to determine how to perform this mapping is the occupancy of the graph. The occupancy of a graph is defined as follows.

Occupancy of a node is defined as the sum of the compute requirements of all tasks assigned to a processor divided by the processor weight.

Occupancy of an edge is defined as the sum of all the flows going over that edge divided by the link capacity of that edge.

Occupancy of the graph is defined as the maximum occupancy over all links and nodes.

How many tasks to map to the processor under consideration is determined based on three criteria.

1. Current Occupancy
   At some intermediate step in the execution of the algorithm, there is a set of mapped tasks and a set of unmapped tasks. The current occupancy of the graph corresponds to the occupancy of the graph calculated based on only the tasks that have already been mapped. This metric ensures that the current processor or links between the current processor and already examined processors are not overloaded.

2. Hypothetical Computational Occupancy
   How many tasks to map onto the current processor depends on what compute power is available on the remaining processors and the compute requirement of tasks that are yet to be mapped. The number of tasks that are assigned to the current processor should be correlated with its relative weight in the entire pool of processors. This metric seeks to ensure load balance among all processors.

3. Hypothetical Outgoing/Incoming Communication Occupancy
   The number of processes mapped to the current processor also depends on the communication capacity between the current processor and the remaining processors. This in effect does not allow a bottleneck to develop in the communication links adjacent to the current processor.

First, the maximum of the three occupancies are calculated given that n remaining tasks are mapped onto the current processor. A list of such values is obtained for different values of n. Then the minimum of the values in the list is computed. The minimum value corresponds to the number of tasks that are mapped onto the current processor (see Figure

4.3). Intuitively, the best number of tasks to map to the current processor corresponds to the number of tasks that does not overburden the current processor or its communication links and also balances the remaining workload yet to be assigned to the set of remaining resources.

Improve Mapping. This step improves the occupancy of the graph by isolating the bottleneck node or edge (the node or edge with the highest occupancy) and removing some tasks from those nodes and re-mapping them. Tasks are selected for re-mapping in such a way that the existing proximity of clusters is maintained. When the re-mapping strategy fails to make any further improvement in the occupancy of the graph, the algorithm terminates.

```
Algorithm Schedule()
{
   Order tasks based on recursive clustering.
   Mapping m = map_tasks();
   do
   {
         improve(m);
   }while(new_occupancy < old_occupancy)
}

Procedure map_tasks()
{
   For every processor
   {
         Find minimum over all mappings possible of the metric:
         Max(Occupancy, Hypothetical Computation Occupancy,
         Hypothetical Communication Occupancy);

         If mapping n of the remaining tasks returns the best
         value for the metric
         Next n tasks are mapped to the current processor;
   }
```

Figure 4.3. Taura's heuristic (GLOBAL)

4.4.2 Hierarchical Algorithm

We describe a new hierarchical, or divide-and-conquer, mapping strategy. There are two levels at which scheduling is performed in this approach.

1. At the granularity of clusters of tasks

   This level of scheduling takes place at the home scheduler when it decides on which zone to execute a cluster of tasks. In this step, a cluster of tasks is mapped onto the set of zones that are available.

   The compute requirements of the tasks in the cluster are aggregated and this represents the compute requirement of that cluster of tasks. Similarly the compute capabilities of the processors in a zone are aggregated and this represents the compute capability of the zone. Communication requirement in the task graph is similarly aggregated for every communication that takes place between tasks in different clusters and communication capacity between zones is the communication delay that is present between zones.

2. At the granularity of a single task
   This level of scheduling takes place at the peer schedulers when they decide the mapping of each task in a cluster to processors in the zone. In this step, each process within a cluster is mapped to individual processors.

At each of these two levels any scheduling heuristic can be used. In the current implementation, we use Taura's algorithm at both levels. In the first level, a node in the aggregated task graph represents a cluster of tasks and a node in the aggregated resource graph represents a zone. In the second level, a node in the task graph represents a single process and a node in the resource graph represents a single processor.

The intuition behind this hierarchical algorithm is to first perform a coarse-grained mapping of task clusters and then refine the assignment within the boundaries of an administrative domain. At each level in the hierarchy different mapping strategies can be used. Some of these mapping strategies are evaluated in Chapter 5. Figure 4.4 shows the pseudocode that is executed at the home scheduler when the application is to be scheduled. The home scheduler determines the level-1-heuristic used and the peer zone scheduler, on which a cluster of tasks gets scheduled, determines the level-2-heuristic used.

Since each cluster of tasks does not require substantial communication with other clusters, the coarse grained mapping is acceptable at the first level of granularity. Also, this results in a more scalable scheduling algorithm.

```
Algorithm ScheduleApplication()
{
        // This step gives the set C of clusters = {$c_1$, $c_2$,…,$c_k$}
        C = ClusterApplication();

        MapTasks(C, Z, Level-1-Heuristic);

        For every cluster $c_i$ mapped to a remote zone $Z_j$
        {
                ScheduleCluster($c_i$, $Z_j$);
        }

        For every cluster $c_j$ mapped to the present zone
        {
                MapTasks($c_j$, $Z_{local}$, Level-2-Heuristic);
        }
}
```

Figure 4.4. Hierarchical heuristic (HIER)

One of the problems present in Taura's heuristic is that the processors are ordered in some sequence and the assignment of tasks takes place from the first processor to the last processor. The sequence in which the processors are ordered can make a large difference in the performance of this scheduling heuristic. This problem arises when a large network distance separates consecutively numbered processors in the distributed system. The tasks in the ordered list of tasks have the property that if they are closer together on the list, then they communicate more intensively than if they are further away in the list. However, when a zone boundary is crossed, the first task placed on the new processor and the last task placed on the previous processor are separated by a large "network" distance. There is no easy way to solve this problem since there are n! ways of ordering the list of processors. The hierarchical mapping strategy mitigates the effect of this problem since every mapping that takes place in this strategy does so at the same level of network distance. Thus at the granularity of clusters of tasks this distance is the WAN network distance and at the granularity of individual tasks, this distance is at the LAN network distance. Thus in this method of mapping processes, the effect of crossing a processor cluster boundary is not very pronounced. However, this effect might still be present if within an organization itself, there are heterogeneous sets of resources. In this case, the strategy could be extended to more than two levels of hierarchy.

The hierarchical scheduling algorithm fits into the ZBS framework as follows. The user contacts the home scheduler with the application and the application model. The home scheduler clusters the application. This results in clusters of tasks. Now each cluster of tasks is given different zones by applying the level 1 heuristic. The tasks within each cluster are then scheduled within these zones. This is achieved by means of contacting the zone scheduler and making a request by invoking the method ScheduleCluster to schedule a group of processes of the application. Each zone scheduler then applies its own level 2 scheduling heuristic and returns the mapping of processes in its zone back to the home scheduler.

4.5 Comparison of Various Scheduling Approaches

Our approach to scheduling has various merits and demerits. In this section, we compare the different methods of scheduling qualitatively. In Chapter 5, we quantify some of these metrics based on experiments.

Merits.

- With the hierarchical strategy, multiple levels of scheduling of the program allows for aggregation of resource information at each level of scheduling. This in turn increases the scalability of the scheduling algorithm.

- The problem formulation is generic and different parameters can be used to generate the weights of the task and resource graph. Thus the method in which these numbers are obtained could be changed without affecting the scheduling mechanisms.

- The scheduling process does not require the intervention of the user due to its lightweight nature. Thus the entire process of scheduling is transparent to the user, which is a necessity for large systems like the Grid.

- The hierarchical scheduling algorithm is coarse-grained and hence the complexity of scheduling is less compared to dependency graph scheduling. Also, scheduling using dependency graphs is not feasible in the presence of administrative domains. The difficulty arises in splitting the dependency graph among domains—each node in the graph represents a block of code within a process and does not represent the entire process. Thus the graph cannot be split and administrative domains can no longer make their own scheduling decisions.

Demerits.

- In order to obtain the task graph, the application processes are characterized based on computation and communication. The actual graph of the application is a time variant graph in which the weights of the nodes (processes) and edges (communication between processes) change over time. In our approach to scheduling, the task graph is obtained by aggregating the application graph over the entire execution time of the application. Thus the scheduling is expected to result in comparatively worse schedules as compared to dependency graph scheduling.

- Since aggregation is performed to give zone schedulers a coarse-grained idea of how to map to a particular zone, this aggregation might lead to loss of information. Aggregation of information about resources within a zone is bad when the resources inside a zone are extremely heterogeneous and hence the aggregate is the result of computing the first moment over a set of resources with high variance in capabilities. If the aggregate value is not representative of the zone, then bad mappings might result.

Chapter 5. Evaluation and Results

Chapters 3 and 4 provided a description of the ZBS framework and how the information required for scheduling is obtained. Chapter 4 also discussed Taura's scheduling algorithm and how we adapted it for use in the zone based framework. This chapter examines two issues. First, we evaluate the efficacy of using the zone based framework as compared to a centralized scheduling strategy. This is used to demonstrate the fact that not only is hierarchical scheduling practical from the point separate of administrative domains, but also that a hierarchical approach can often produce better mappings than centralized solutions. Secondly, this chapter describes experiments designed to evaluate how good the representation of the task and resource structure is for scheduling purposes and how changes could be incorporated in the representation to improve the mappings obtained. One point that has been stressed in this thesis is that the formulation that we use (the resource-task graph) can be obtained in many different ways. We used a simple method to obtain resource and task graphs, but we demonstrate in this chapter that making even simple changes to take into account more factors leads to better mappings. This is important for future development of the scheduler and demonstrates that such a representation, while both simple and easily obtained, can in fact be made very realistic by taking more factors into account.

The experimental results obtained from three sets of experiments are described in this chapter. Section 5.1 provides the basis of comparison of scheduling algorithms and also describes the algorithms that are compared. Section 5.2 introduces the class of applications that motivate our approach to scheduling and describes the performance of one specific application. It also describes a synthetic test suite that was designed for these experiments. Section 5.3 describes the simulation environment. Sections 5.4 and 5.5 describe the results obtained from the simulation study.

5.1 Introduction

The main aim of scheduling is to reduce the time from which an application is given to the system to the time at which results of the application are obtained. There are three components to this turnaround time of the application. Each of these three aspects of the scheduling of parallel programs on the Grid was compared. Each aspect was evaluated on a different set of experiments due to the difficulty in correlating these three areas. The methods used to evaluate each of these facets of a scheduler are described in Sections 5.4–5.5. The three aspects to minimizing job completion time are:

1. **Information Gathering Overhead.** In order to perform scheduling in an application-aware and resource-aware manner, information about the application and resource has to be collected. Different strategies for obtaining this information have different overheads. This information-gathering overhead is evaluated in Section 5.5.1.

2. **Scheduling Overhead.** Given information about the set of tasks and a set of resources, the scheduler makes a decision about which tasks to place on which

resources. The time required for scheduling depends on the sophistication of the scheduling algorithm and on the size of the set of tasks and resources. Scheduling overhead is evaluated in Section 5.5.2.

3. **Running Time and Related Metrics.** Once the mapping of tasks to resources has been performed, the application needs to be run on the resources identified based on the mapping performed. The qualities of schedules obtained from various scheduling algorithms and for different applications and resource structures are evaluated based on running time in Section 5.4.

Different scheduling algorithms were compared on the three aspects mentioned above. The two methods of scheduling being compared are described in Chapter 4. They are:

- Global Mapping (GLOBAL)

  In this method there is a single global scheduler that has complete information about all the resources and all the tasks. The global scheduler uses Taura's algorithm to perform the mapping.

- Hierarchical Mapping (HIER)

  In our experiments, we apply Taura's algorithm at both levels of granularity (at the granularity of a single task and at the granularity of a cluster of tasks). Note that this is one specific instance of a hierarchical scheduling algorithm; any algorithm could be used at each level.

5.2 Evaluation Test Suite

The evaluation test suite consisted of two sets of applications. The first application is a test kernel that resembles a large parallel SPMD application. The second application is a parallel program that generates communication between random pairs of processes, with a structure superimposed on this random communication. The characteristics of these programs are described in this section.

5.2.1 TRANSPORT Kernel

The ZBS scheduling framework has been designed to efficiently support a certain type of application, namely, applications whose task graphs have highly connected sub graphs. The application TRANSPORT is our main motivating application falling under this general class of applications. TRANSPORT is an MPI based application used to calculate the conductance properties of various molecules in various orientations. The main property of interest to us is the computation-communication structure of this application that makes it suitable for a distributed scheduling framework. It is a master-slave application in which the master allocates computations corresponding to different "energy levels" to the different slaves. The slaves perform their work and return the result to the master. Each slave is not one single entity but actually a *team* of processes interacting with each other to solve sets of linear systems of equations. The linear systems of equations are solved in parallel by means of parallel solvers such as ScaLAPACK [49]. Each team of processes that are involved in solving a linear system of equations in parallel communicates a great deal among themselves. However, there is not much interaction between such teams.

TRANSPORT is a parallel application having large computation, communication and memory requirements. In order to be able to test an application like TRANSPORT and applications with a similar structure on machines with limited processing power and memory, a synthetic test suite kernel that resembles TRANSPORT in its characteristics was designed and implemented. This test suite has smaller memory requirements and running time as compared to TRANSPORT but is otherwise very similar in its computation-communication structure. In addition, this synthetic test kernel allows us to vary the application's execution in a controlled manner, as compared to the variance in TRANSPORT which is based on numerous problem parameters that are not easily manipulated to simulate different application characteristics.

The test kernel resembles the master-slave team structure of TRANSPORT by means of a parameter file as follows (see the example in Figure 5.1). NO_OF_TEAMS represents the number of parallel solvers that are involved and NO_OF_PROCS_PER_TEAM represents the number of processes within a single team. There are parameters that specify how many bytes are communicated within a single team and between teams for every communication operation (BYTES_INTRA_TEAM and BYTES_INTER_TEAM). The MPI operation performed between the master and slave team leader (MPI_Bcast in Figure 5.1) and the MPI operation within a slave team (MPI_Allgather in Figure 5.1) can also be varied. The NO_OF_INTERNAL_CYCLES represents the number of within-team communication operations taking place for every single external operation performed. TIME_TO_COMPUTE represents, in terms of the number of *noop* instructions, the computation time between communication operations.

```
NO_OF_TEAMS=2
NO_OF_PROCS_PER_TEAM=2
BYTES_INTRA_TEAM=1000
BYTES_INTER_TEAM=1000
OP_INTRA_TEAM=MPI_Allgather
OP_INTER_TEAM=MPI_Bcast
NO_OF_EXTERNAL_CYCLES=10
NO_OF_INTERNAL_CYCLES=1000
TIME_TO_COMPUTE=100000
```

Figure 5.1. Typical parameter file used in the test kernel.

The communication structure of the test kernel can be varied by means of the parametric information stored in the parameter file. The underlying structure of the parameter file is very similar to the communication structure of TRANSPORT. Table 5.1

compares the number of bytes of communication between the TRANSPORT application (left) with the synthetic application (right). Both these matrices represent 5 processes (1 master and 2 teams of slaves each having 2 processes). TRANSPORT's communication matrix has processes (1,3) and (2,4) being the slave teams whereas in the synthetic application (1,2) and (3,4) are the slave teams (note the large values in these cells).

	0	**1**	**2**	**3**	**4**
0	0	28322	28322	26376	26688
1	18208	0	212	72e+6	378
2	18126	212	0	496	73e+6
3	16448	62e+6	312	0	212
4	16762	192	61e+6	212	0

	0	**1**	**2**	**3**	**4**
0	0	6156	396	5760	236
1	236	0	400000	236	0
2	236	400000	0	396	0
3	0	236	236	0	400000
4	236	0	0	400000	0

Table 5.1. TRANSPORT communication structure (left) and synthetic application communication structure (right).

5.2.2 Randomly Communicating Program

In order to simulate applications with more diverse communication patterns than TRANSPORT, another set of MPI based parallel programs was written. Each process in these SPMD programs computes for some time and then randomly communicates with another process. In order to introduce clusters into the communication structure of the program, the user inputs the number of clusters to be present in the application. Using the random() function, each process in the application is assigned to a particular cluster. Once the set of clusters is identified, the communication is determined so that there is much more communication between processes in the same cluster than communication between processes in different clusters. Transmitting only every i^{th} message between processes in different clusters and every message between processes in the same cluster accomplishes the desired pattern of communication.

These random communication programs allow us to simulate multiple levels of communication intensity, as compared to the TRANSPORT kernel, which has only two very distinct levels of communication intensity. It also allows us to vary the message frequency and size, and to vary the amount of computation between communication events. The goal is to generalize the results obtained from the TRANSPORT kernel to more generic applications that perform a sequence of computations and communications.

5.3 Simulation Environment

This section describes how the experimental results presented in Section 5.4 were obtained. The set of applications described in the previous section were run on the Anantham parallel cluster. The Anantham cluster consists of 200 AMD Athlon 1Ghz processors, each with 1 Gb of memory, and interconnected by a 2.56 GB/s Myrinet network. A set of log files is obtained as a result of running these programs – one log file from each process. These log files have entries recorded as described below.

There are two points where logging information is written:

1. The parallel application invokes TCP's read, readv, write and writev function calls present in the glibc library to send or receive data. A set of wrapper routines that mimick these calls' function signatures was created. These wrapper routines are linked with the parallel application. Whenever the TCP routines are called, these wrappers intercept these calls before the actual TCP routines are called. These wrapper routines log into a file timestamps whenever a read from or write to a network buffer occurs.

2. Additional code was written in MPICH's functions (more precisely, MPI_Send, MPI_Recv, MPI_Isend, MPI_Irecv and MPI_Waitall functions) to record the timestamps at which these functions are invoked. The MPICH p4 device was used as the MPI implementation on which experiments were carried out.

The instrumented MPICH functions produce one log file for every process of the parallel program. Each process writes to a log file that is uniquely identified by its processor name. This log file provides the execution trace of the parallel program when running on a processor cluster. This execution trace is processed and fed into a simulator (the *dlsim* simulator was modified to use these log files). The processing of the log files of the various processes involved matching up the send from one process to a receive on its counterpart process and finding out the time difference in their timestamps. The clocks on Anantham are synchronized by means of NTP (Network Time Protocol). This log file processing step gives the time to send each message for the Anantham cluster. We are able to simulate the communication operation for networks with different network parameters by scaling these message times appropriately.

The end-to-end delay between processors is given to the simulator. Based on the execution trace of the parallel application, the application is simulated to run over a network with different end-to-end delays and different processor speeds. The computation part of a parallel program is the time between two successive calls to MPI_Recv or any variant of MPI_Wait. This is an acceptable way to model the blocking behavior of parallel programs since all collective communications are reduced to point-to-point operations inside MPICH. The blocking time at a particular process is the difference between the time that program called MPI_Recv or any variant of MPI_Wait and the time the actual data arrived. If a network with longer delays is simulated, the process will block for a longer period of time thereby increasing execution time.

5.4 Evaluation of Scheduling Heuristics

This section seeks to answer the following questions:

- Does the distributed scheduler, having aggregated and hence incomplete information about resource structure, produce competitive schedules as compared to a global scheduler that has complete information about resource structure?

- What is the best size for clusters in the task graph when the home scheduler splits the application into clusters?
- How does frequency of messages affect the schedule obtained?

The experimental results obtained are shown in this section. The results are obtained from executing the applications described in Section 5.2.

5.4.1 Comparison of Global and Hierarchical Scheduling Policies

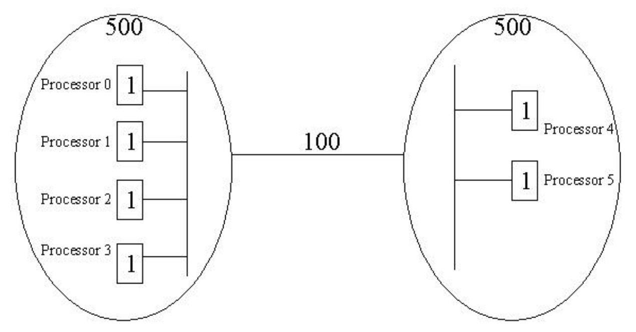
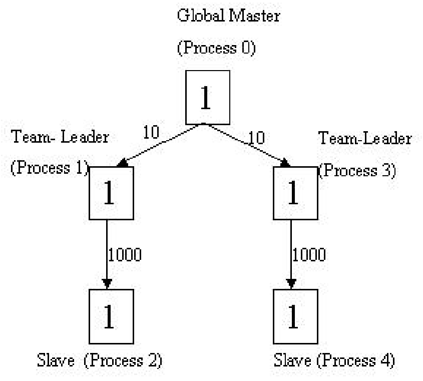
Resource Graph	Task Graph
 500 500 Processor 0 Processor 1 Processor 2 Processor 3 Processor 4 Processor 5 1 100	 Global Master (Process 0) 1 Team- Leader (Process 1) 10 10 Team-Leader (Process 3) 1000 1000 Slave (Process 2) Slave (Process 4)

Figure 5.2. A simple resource and task graph.

We first compare the schedules obtained by the GLOBAL and HIER algorithms on a simple example. The comparison is based on the running time of the TRANSPORT kernel application with the task graph shown in Figure 5.2 (right), and run on the two zone resource graph shown in Figure 5.2 (left). In this experiment, the nodes in both the task graph and the resource graph are equally weighted. Thus, only communication is the determining factor in the task assignment. Also, the zones are homogeneous in that they have the same interconnection speed (500 in Figure 5.2) and same processor speed (1 in Figure 5.2). The mappings obtained from the two scheduling algorithms are evaluated using the simulator with the parameter file shown in Figure 5.1.

The mappings obtained from the different mapping strategies are given in Table 5.2. The first number in the order pair represents the processor and the second number represents the process mapped onto that processor.

GLOBAL	{(0,0),(1,1),(2,2),(3,3),(4,4),(5,-)}
HIER	{(0,0),(1,1),(2,2),(3,-),(4,3),(5,4)}

Table 5.2. Mappings obtained from GLOBAL and HIER.

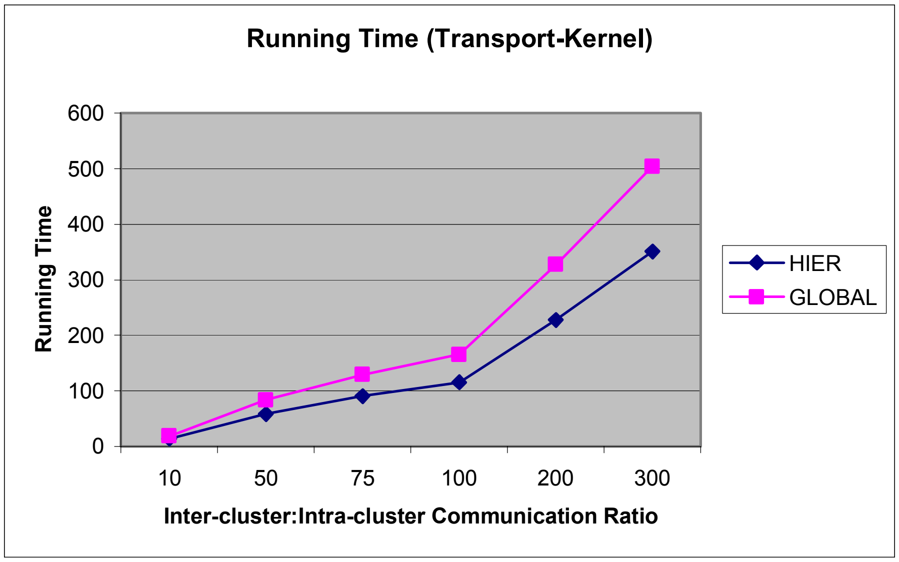


Figure 5.3. Running Time of TRANSPORT Kernel.

Figure 5.3 compares the performance of schedules obtained from the GLOBAL and HIER mapping strategies. The x-axis varies with the ratio of NO_OF_INTERNAL_CYCLES to the NO_OF_EXTERNAL_CYCLES. As can be seen from the results, the running time obtained from the hierarchical mapping strategy is better compared to the global strategy since the problem of processor cluster boundaries being crossed during the mapping process is mitigated (this problem is described in Section 4.4.2). As the ratio of internal to external communication increases the application mapped by GLOBAL performs progressively worse than the mapping from HIER.

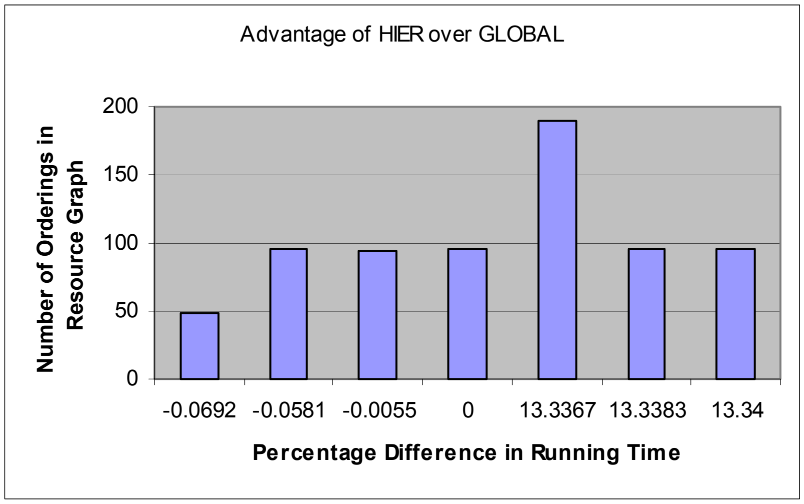


Figure 5.4. Advantage of HIER over GLOBAL for all possible resource graph orderings.

Since the ordering of resources in the resource graph is an important consideration in Taura's algorithm, we wanted to check that we had not unfairly biased the GLOBAL strategy. In order to do this we took all possible (6!) orderings of the resource graph and compared the number of times HIER beat GLOBAL and by how much. This was for the case when the ratio of inter to intra cluster communication is 100, where the total execution time was approximately 60 seconds. From Figure 5.4 we notice that there are 380 cases in where HIER is better, and 332 cases when the performance is essentially the same. This advantage of HIER over GLOBAL will be more pronounced when the difference in network distance (end-to-end latency) between zones is larger. The results suggest that the HIER scheduling strategy consistently obtains a close to optimal mapping, while the GLOBAL strategy has a substantial risk of choosing a clearly sub-optimal mapping.

5.4.2 Effect Of Message Size and Frequency

Our approach to application modeling aggregates communication (by summation) and computation (by averaging) over all communication and compute steps. While this provides a simplistic model of the parallel application, one that can be easily scheduled in an efficient and distributed manner, it also results in a loss of information that might be vital to the performance of the scheduler.

In the original representation of the application, we used total amount of communication between each pair of processes to determine coupling of processes. Recall that a communication matrix can be used to represent the edges between tasks in the task graph; a communication matrix is an adjacency matrix weighted by amount of communication. If two entries in the communication matrix have the same values, they are considered to be coupled with the same intensity during the formation of clusters. However, one intuition that is verified by means of the experiment described in this section is that even if aggregate values in the communication matrix are the same, the process pair that has a greater number of messages exchanged should be considered more tightly coupled. The intuition is that more (smaller) messages implies more synchronization points between that pair of processes, and hence tighter coupling.

The randomly communicating program was used in this experiment to verify the advantage of incorporating number of messages into the application model. There were three clusters of tasks, identified by C0, C1 and C2, with each cluster having two tasks. Communication between C0 and C1 consists of many smaller messages, whereas communication between C0 and C2 consists of a few large messages. In the experiment, C0 sent 10 messages to C1 of size 100 bytes for every 1000 byte message sent by C0 to C2. Thus processes in C0 and C1 communicate 10 times more frequently than processes in C0 and C2. However, the messages between processes in C0 and C1 are 10 times smaller that messages between processes in C0 and C2. Thus the aggregate communication between processes in C0-C1 and C0-C2 are equal.

The skewing of communication is done as follows. The source and destination process pairs are generated randomly. This pair of source and destination processes is represented in one line in a "communication pattern file". The pattern file contains many such lines of randomly generated source and destination process pairs. The parallel program reads each line in this file and if the source and destination are within the same cluster, a message is sent every time such a line is read from the file. If *i* is the number on x-axis in Figure 5.5, then this implies that every *i*th line read from the communication pattern file results in a message sent between processes in clusters C0 and C1 and every *i**10th line read results in a message sent between processes in clusters C0 and C2. For instance, the first point on the x-axis corresponds to: "if source is in C0 and destination is in C1 then every 10th time a short message is sent and if source is in C0 and destination is in C2 then every 100th time a long message is sent".

The resource graph for this experiment had three zones. Each zone had four processors each of equal processor speed. The end-to-end latency between zones was four times slower than the end-to-end latency within a single zone.

A comparison was performed in which scheduling is performed taking the number of messages into account and not taking number of messages into account. The number of messages was taken into account by weighting the cells in the communication matrix that have higher number of messages by a larger amount. For this experiment, we multiplied the cells in the communication matrix between C0 and C1 by 2 and the cells in the communication matrix between C0 and C2 by 1.

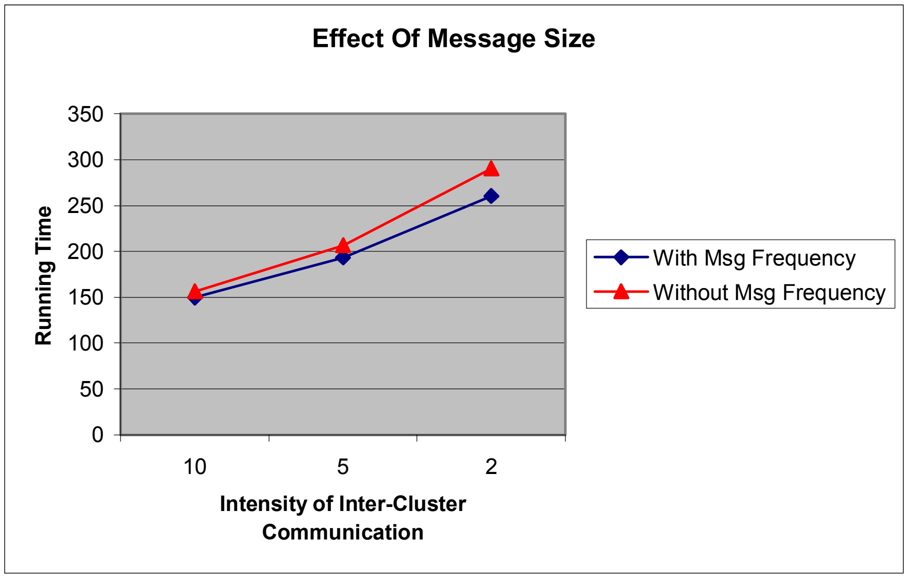


Figure 5.5. Comparison of scheduling strategies taking message frequency into account.

Taking the number of messages into account the schedule obtained was found to be better than not taking them into account while clustering the application. Also the difference becomes more apparent in applications that are communication intensive. This is observed by the increase in the difference in the two sets of plots as the intensity of communication increases along the x-axis in Figure 5.5, i.e., as the relative amount of inter-cluster communication increases.

5.4.3 Effect of Cluster Size

Recall that clustering is done in the experiments described here using the algorithm designed by Yeh et al. This algorithm works by injecting flows into the task graph till the clusters are formed. The mapping algorithm designed by Taura et al. applies this algorithm recursively. One of the issues when performing clustering at the home scheduler is to determine at what granularity the clusters should be to allow for effective mapping to the various zones. This is an issue that is specific to the HIER strategy and does not arise in the GLOBAL strategy. In the GLOBAL strategy, the clustering is performed recursively till singleton elements are obtained. However, in the HIER strategy, the clustering should stop at some level of clustering before singleton elements are obtained. This is required to have the benefits of the divide and conquer strategy present in HIER, since now the clusters of tasks can be mapped onto zones, and the larger the clusters the less the possibility of zone crossover and the faster the scheduling is performed.

TRANSPORT has a well-defined two level hierarchy in intensity of communication operations. Thus splitting a team across wide area networks is not good and is easily identified by the clustering algorithm. However, there may be parallel programs in which the change in communication intensity is more gradual and in which three or more levels in communication intensity are present. The different levels of clustering are identified by means of the tree of clusters in Taura's algorithm (refer to Section 4.4.1). When performing the mapping at the granularity of zones, the clusters at a certain level must be chosen to correspond to the nodes in the aggregated task graph. Thus a method to determine at which level clustering should be stopped is required. Stopping the clustering at a very high level might result in very coarse clusters whose computation may not be handled by the zones on which they are scheduled. On the other hand, making these clusters very fine might result in large communication across zones and hence large wide area traffic.

Consider a resource graph with two zones. The 1st zone has 6 processors and the 2nd zone has four processors. All the processors are equally weighted and the end-to-end latency between zones is four times slower than end-to-end latency within a zone.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0	140556	3596	3520	1484	1408	1408	1408
1	134844	0	3520	3772	1408	1484	1408	1408
2	3596	3520	0	144044	1408	1408	1660	1408
3	3344	3596	132220	0	1408	1584	1408	1484
4	1692	1408	1408	1584	0	145804	3596	3520
5	1408	1484	1584	1408	134012	0	3520	3596
6	1408	1408	1484	1408	3596	3696	0	133836
7	1584	1584	1584	1484	3520	3596	136300	0

Table 5.3. Communication matrix for the synthetic application.

Three levels of communication hierarchy were simulated by means of a parallel program that randomly sent and received messages between pairs of processes. There were 8 processes, with the communication matrix shown in Table 5.3 (recall that the communication matrix is an adjacency matrix weighted by amount of communication).

From the communication matrix given above, we notice that there are three different intensities of communication. Thus the cluster levels are:

Level 1: (0,1,2,3,4,5,6,7)
Level 2: (0,1,2,3) (4,5,6,7)
Level 3: (0,1) (2,3) (4,5) (6,7)

If processes are related at a higher level they have a greater intensity of communication. Thus processes 0 and 1, which are related at level 3, communicate a lot more as compared to 0 and 2, which are only related at level 2. The skewing of communication is performed in a similar manner to the previous experiment. The source and destination processes are generated randomly and written to a "communication pattern file". If processes are related at level 3, then every message is sent; if processes are related at level 1, every 100th message is sent. The number on the x-axis of Figure 5.6 represents the difference in intensity of communication between level 2 and level 3. Thus the points on the x-axis represent that for processes related at level 2, every 20th, 10th, 5th and 2nd messages is sent, respectively. In other words, intensity of communication at level 2 increases from left to right in the figure.

The decision to be made is at what level to stop the clustering algorithm. In this simple example, we can either stop with coarse-grained clusters at level 2 or more fine-grained clusters at level 3. Figure 5.6 shows that fine-grained clusters are better when the difference in communication intensity between levels 2 and 3 is large. As communication at level 2 increases, splitting the clusters into fine clusters makes it more likely that processes in the same cluster in level 2 will be split over wide area networks, which makes coarse-grained clusters better. This can be seen by the reduction in performance of using fine-grained clustering with increasing communication along the x-axis.

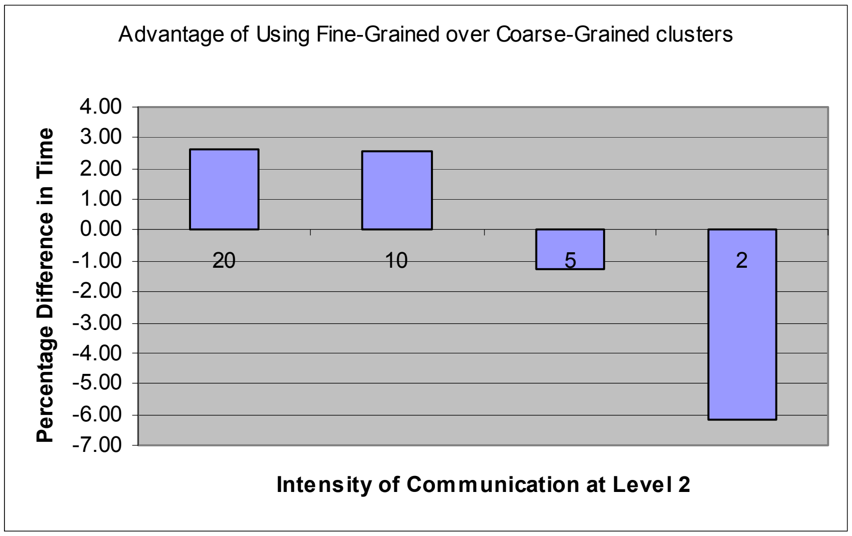


Figure 5.6. Effect of cluster size on performance

The tradeoff between node occupancy and edge occupancy determines at which level to stop the clustering. In this paragraph, a node refers to a zone and edge refers to the link between two zones. If coarse clusters are obtained, the node occupancies of some of the zones will be large, since they will not be able to handle such a great computational requirement. If fine clusters are obtained, the edge occupancies of some of the links will be large due to splitting clusters across zones. Thus nodes become the bottleneck with coarse clusters and edges become the bottleneck with fine clusters. One heuristic that could be used is to proceed to the next level of clustering only if the performance degradation due to increased edge occupancy does not offset the improvement in node occupancy.

5.5 Evaluation of Overheads

This section seeks to answer the following questions:

- What is the overhead involved in getting information about load of processors and networks transmitted from the location where it is monitored (i.e., the resources) to the location where it is used (i.e., the scheduler) for the different methods of scheduling?
- What is the trend in scheduling time as the number of processes and processors increases for different methods of scheduling?

5.5.1 Information Gathering Overhead

The loads on processors and networks change based on the number and type of applications currently running. The current and forecasted load play a major role in determining where the application should be scheduled. This information has to be transmitted from the sensors running on the machines to the place where scheduling takes place. In this section, we compare the quantity of resource information transmitted for the centralized (GLOBAL) and distributed (HIER) scheduling strategies.

In order to compare the amount of information transferred for different methods of scheduling, it is necessary to determine certain criteria on the basis of which a comparison can be made. Information collected to make a single scheduling decision for a single application forms this basis of comparison. Information gathering systems like MDS and NWS populate information periodically into their repositories. This periodicity of information gathering is not taken into consideration. However, periodicity only increases the difference in the amount of information transmitted between GLOBAL and HIER scheduling strategies.

In the global scheduling strategy (GLOBAL), the number of units of information going over WAN links for scheduling on *z* zones with P_i processors in the ith zone is calculated as follows. Each zone has information about network load from every processor in the current zone to every processor in other zones. Thus the amount of information stored at each zone is proportional to,

$$\sum_{i=1}^{z-1} P_i^2$$

This information is transmitted to the central location at which scheduling takes place. Thus each of the (z-1) zones transmits this information and the information about the load of processors in its zone. Thus the total amount of information transmitted between zones is

$$(z-1)\sum_{i=1}^{z-1} P_i^2 + \sum_{i=1}^{z-1} P_i$$

In the distributed scheduling strategy (HIER), the number of units of information going over WAN links for scheduling on z zones with P_i processors in the ith zone is

(z) $(z-1)$

This is obtained since each zone provides one unit of information to every other zone. This one unit of information is representative of the resources in that zone. This might be more than a single unit of information if more advanced information is to be provided, such as variance. In the current implementation of the scheduler we do not use variance and hence the above equation results. Also, the number of units of information propagated will not be proportional to the number of processors in the zone and hence will be a constant multiplicative factor that does not affect its asymptotic order.

The amount of information transmitted over wide area networks for GLOBAL is bounded by $O(z^2 * P_{max}^2)$ (where P_{max} is the number of processors in the largest zone) whereas in the HIER scheduling the amount of information is bounded by $O(z^2)$. Thus as the number of processors within a single zone increases, the hierarchical strategy wins by larger and larger amounts.

5.5.2 Scheduling Overhead

One of the advantages of a divide and conquer approach to scheduling, as described in this research, is that the time required to make scheduling decisions is reduced. The experiment described in this section quantifies the scheduling time for different scheduling strategies. Task and resource graphs were given as input to the different scheduling mechanisms. To simulate large graphs, random task and resource graphs were generated with differing number of clusters and different graph sizes. The schedulers were run on a 2 processor SMP with combined processor speed of 2 Ghz.

The running time of the two algorithms was measured and plotted as shown in Figure 5.7. Ignoring communication and queuing delays in transmitting the application profile to the remote schedulers, the hierarchical scheduling strategy is compared with the global scheduling strategy. The x-axis of this figure represents the total number of tasks (T) multiplied by the total number of processors (P) for which the scheduling decision is made and the y-axis represents how long the scheduling took to perform. As can be seen from the figure, the hierarchical (HIER) mapping strategy takes much less time to compared to the global (GLOBAL) mapping strategy.

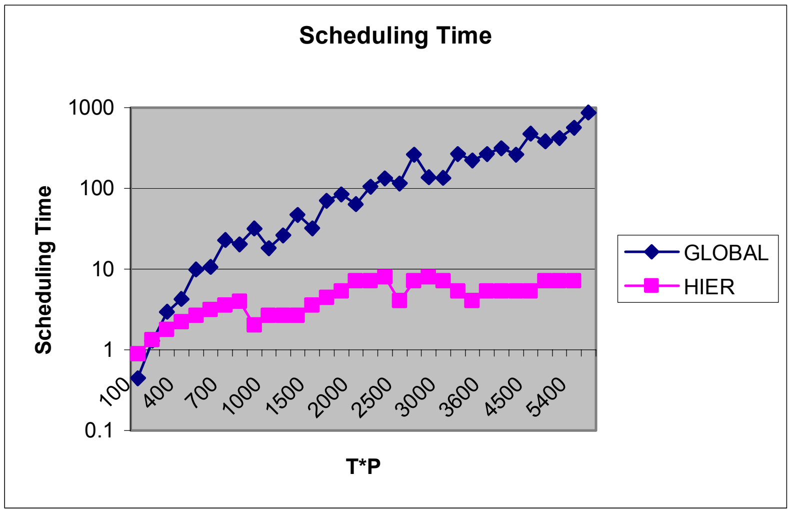


Figure 5.7. Scheduling time for GLOBAL and HIER scheduling strategies.

5.6 Evaluation Summary

The GLOBAL and HIER scheduling strategies have been compared on the basis of the mappings they give for the TRANSPORT kernel. As the ratio of the inter-cluster to intra-cluster communication intensity increases, we see that the HIER strategy is more likely to perform better than the GLOBAL strategy. This effect is due to the greater negative impact of separating highly communicating processes over wide-area links.

The adequacy of representing the application by means of average compute time between communication operations and number of bytes of communication is evaluated. From the experiments, it is seen that while a simple mechanism of representing the application is often sufficient, incorporating additional elements like variance in computation steps and message frequency into the representation allows for a better mapping of tasks. This suggests possibilities for further investigation of better metrics to determine compute and communication intensity of processes.

The GLOBAL and HIER strategies have also been compared on the basis of the overhead in using each of those strategies. The amount of information to be collected about resource structure for HIER is predicted to be significantly less than the amount of information collected in GLOBAL. Also, due to the divide and conquer nature of HIER, it is shown to perform scheduling in less time compared to the GLOBAL strategy.

Chapter 6. Conclusions and Future Work

6.1 Conclusions

We have designed and implemented a zone-based scheduling framework as a means to provide site autonomy for different organizations while at the same time providing acceptable schedules to applications. The scheduling framework developed is lightweight and extensible. The framework addresses security problems of having to provide user account information to the scheduler, and the aggregation of resource information obviates the need for complete internal information about an organization’s resources to be transmitted to a scheduler outside the organization. The decentralized scheduling algorithm used in this research offers a divide and conquer solution to the scheduling problem. This makes both the scheduling and the information gathering services scalable.

We have also used a simple method of representing applications and evaluated scheduling algorithms using this representation. While this is a lightweight approach to scheduling, the application and resource models used by the scheduler are simplistic. The reason for using simple metrics is that the environment on which the application is executed should not influence these metrics. It is important to make use of those factors that are constant and unlikely to provide misleading information about the application’s structure. We use basic metrics like the average number of cycles between communication operations and number of bytes of communication.

6.2 Contributions

A Cross-Organization Collaborative Framework. The main contribution of this thesis is in the identification and implementation of a scheduling framework under which policies of distinct administrative domains can be enforced. In the ZBS framework each organization can collaborate with others while at the same time maintain full control of its resources. Thus, administrators in an organization can enforce policies (some of which might be temporary) on usage of resources while scheduling. Also, newer scheduling mechanisms can be incorporated into a scheduler in a manner that is transparent to the user.

Dynamic Virtual Organizations. A method to facilitate easy formation of virtual organizations using UDDI registries to discover organizations and their resources has been demonstrated. Once these organizations have discovered each other, a means by which they can effectively collaborate has been demonstrated.

Comparison of Scheduler Implementations in the Framework. Scheduling algorithms implemented in this framework show acceptable performance and result in an improvement over the base algorithm (Taura’s algorithm). Also, certain interesting issues that occur with the hierarchical scheduling strategy and possibilities of better application representation methods are examined.

Information Gathering about the Application. The author has also had to make modifications to the way information about the application is obtained in MPI's profiling interface. When an application runs in a grid environment, it is difficult to apply information obtained from one run to the next run. Simple application specific metrics independent of the resources on which the application runs have been identified and modifications to the execution environment (MPICH) have been performed to be able to obtain this information.

6.3 Future Work

Interaction with Other Grid Middleware. One interesting possibility for the ZBS framework is to consider its interaction with other grid middleware, such as runtime algorithm selection systems. A combination of these two systems appears to be beneficial and has been explored by Bora et al. [45].

In Figure 6.1, we see the typical development cycle of high-end computational programs. In order to improve performance of a particular code, the code is run, performance data is obtained and analyzed. Then the code is rewritten to improve performance, either by using a different algorithm, or by changing parameter values or by choosing a different set of machines. In an effort to take the burden of trying different combinations of algorithms, parameters and machines off the application scientist, there is a need for middleware that will automate the process. We believe this middleware should include a runtime algorithm selection system and a zone-based scheduler. Thus, in Figure 6.1 (bottom), choosing a solver from a set of libraries and tuning the associated parameters, is done by the runtime algorithm selection system. However, this changes the sequence of operations performed to improve the performance of the application. The scheduler can no longer make intelligent decisions about which resources to select for the application as this information is encapsulated in the runtime algorithm selection system. Thus there is a need for these two components to interact with each other.

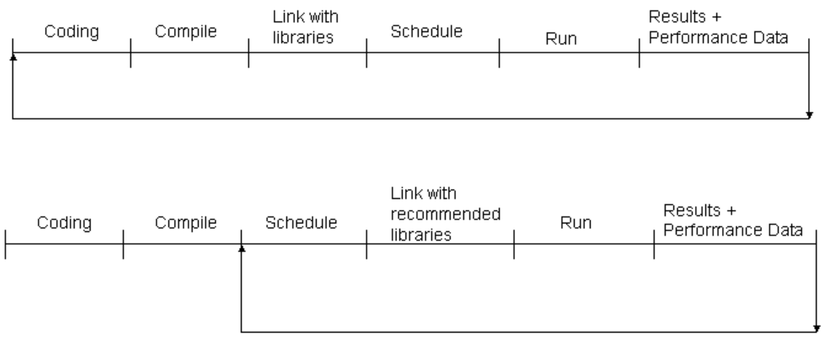

Coding
Compile
Link with libraries
Schedule
Run
Results + Performance Data
Coding
Compile
Schedule
Link with recommended libraries
Run
Results + Performance Data

Figure 6.1. Performance improvement cycle for an application: without algorithm selection (top), with algorithm selection (bottom)

A promising interaction scenario is as follows. The scheduler queries the runtime algorithm selection system to obtain information about what type of resources are best for the particular application. Using this information the search space of the scheduler will be reduced to only those machines that can provide good performance for that application. A typical interaction is depicted below. From prior runs of an application, the runtime algorithm selection system finds out performance information that is stored in a table such as this:

	Machine Characteristics	Algorithm Selected	Performance Data
Class 1	{Pentium II, cache size 1, memory size 1}	A1	Time 1
Class 2	{Alpha, cache size 2, memory size 2}	A2	Time 2

Table 6.1. Records exchanged between runtime algorithm selection system and zone-based scheduler

The two rows shown above represent the performance data of an application. This table is sorted by performance. Thus, the application when run on Pentium II machines with cache size 1, memory size 1 and algorithm A1, performs better as compared to the second combination (Class 2) of machine type and algorithm. (For now, we assume "best performance" means lowest execution time.) Although the scheduler is not interested in the algorithm selected, the characteristics of the machines on which the application performed well is of importance to it. In the above example, the application will perform better if it is scheduled to run on "Class 1" machines. Thus the scheduler will attempt to choose machines with those characteristics when assigning tasks to processors.

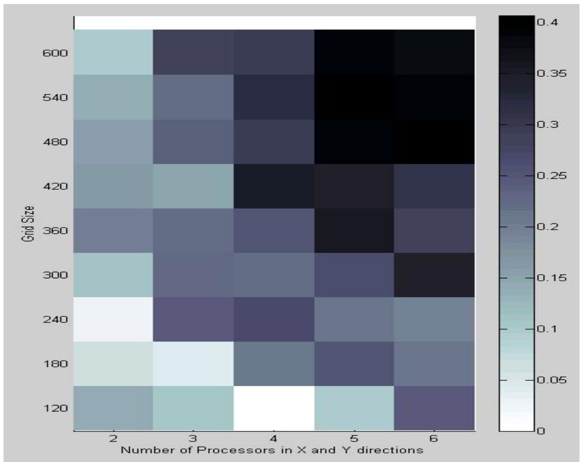


Figure 6.2. Relative performance penalty of using GMRES restart value 20 instead of the optimal value, as a function of problem size and number of processors. Matrix generated from finite difference discretization of test PDE problem 15 from [47].

As a simple example of the benefits of combining ZBS with runtime algorithm selection, consider the relatively simple algorithm-selection problem of choosing the parameter k (the size of the Krylov subspace) for a parallel restarted GMRES algorithm. The value of k can have a significant effect on the performance of the algorithm; and more importantly, the optimal choice for k depends in subtle ways on the problem, the amount of memory available, the CPU speed, and the interconnection network. In practice, most users simply set k to a constant value and proceed. However, the data in Figure 6.2 shows that even for a modestly large problem, a better choice for k can improve running time by 40% or more. Per-processor memory availability is an important determinant in choosing the value for k since a larger value of k requires more memory. The result of the interaction that takes place between the runtime algorithm selection system and the zone-based scheduler is that the scheduler assigns a higher weight to processors that have larger memory available and hence permit a higher value of k. This will allow better performance for applications involving the GMRES algorithm.

Coexistence with Other Grid Schedulers. In this project we have evaluated a small set of scheduling algorithms. However, there is a large base of schedulers already present that control clusters of processors. Integration and evaluation of such resource level schedulers within zones should be evaluated in the context of the zone based framework.

Scheduling Multiple Applications. The scheduling algorithms implemented and evaluated in this project try to obtain a good mapping for an application given the current state of resources. However, in grid settings, it would be expected that there are multiple competing applications. While the zone-based framework is generic and mechanisms to deal with race conditions could be evaluated for such systems, this was beyond the scope of the project.

Trust Model. When a scheduler makes a request to register with another scheduler, the answer returned is boolean, i.e., the other scheduler answers yes or no based on whether it trusts users in that organization. While this provides a straightforward implementation, there might be a need for a better way to develop trust between organizations. A model of *degree of trust* between organizations could be developed in such a way that as organizations interact more with each other their level of trust in the other increases.

Application Representation and Performance Models. In the scheduling algorithms evaluated in this project, we used the resource-task graph formulation of the scheduling problem. Feasibility and efficiency of other representations of the application needs to be evaluated in the context of the ZBS framework. Also for long running programs, the log files obtained will be large. This way of obtaining information about the application in the ZBS framework can be improved by modifying MPICH code to record only the summarized information in the log file, i.e., the average computation requirements of a process and bytes of communication between processes.

Further Automation of Scheduling Process. In the current implementation, the user requests for the application to be scheduled. The home scheduler returns the mapping to the user. The user then runs the application (refer to Figure 3.1). The step in which the home scheduler returns the mapping to the user and the user runs the program using this mapping (optional user interaction 2 in Figure 3.1) can be combined with the previous interaction to schedule the program (user interaction 1 in Figure 3.1). Also, in an effort to further reduce the burden on the user, the performance information collected after the execution of the program could be stored in a database maintained at the home scheduler. This would automatically be used in the next run of the program.

Bibliography

[1] J.M. Schopf. **Ten Actions When SuperScheduling**, GGF I.4 document.

[2] H.Dail, F. Berman, H. Casanova. **A Modular Scheduling Approach for Grid Application Development Environments**, to appear in Journal of Parallel and Distributed Computing (JPDC), 2003.

[3] F. Berman and R. Wolski. **Scheduling from the perspective of the application.** Proceedings of High-Performance Distributed Computing Conference, 1996.

[4] K. Kennedy, M. Mazina, J. Mellor-Crummey, K. Cooper, L. Torczon, F. Berman, A. Chien, H. Dail, O. Sievert, D. Angulo, I. Foster, D. Gannon, L. Johnsson, C. Kesselman, R. Aydt, D. Reed, J. Dongarra, S. Vadhiyar, R. Wolski. **Toward a Framework for Preparing and Executing Adaptive Grid Programs.** Proceedings of NSF Next Generation Systems Program Workshop (International Parallel and Distributed Processing Symposium 2002), Fort Lauderdale, FL, April 2002.

[5] S. S. Vadhiyar, J. J. Dongarra. **A Metascheduler for the Grid.** HPDC-11, the Symposium on High Performance Distributed Computing, July 2002, Edinburgh, Scotland.

[6] Dennis Gannon, Randall Bramley, Geoffrey Fox, Shava Smallen, Al Rossi, Rachana Ananthakrishnan, Felipe Bertrand, Ken Chiu, Matt Farrellee, Madhu Govindaraju, Shriram Krishnan, Lavanya Ramakrishnan, Yogesh Simmhan, Alek Slominski, Yu Ma, Caroline Olariu, Nicolas Rey-Cenevaz. **Programming the Grid: Distributed Software Components, P2P and Grid Web Services for Scientific Applications.** Journal of Cluster Computing, 2002.

[7] S. J. Chapin, D. Katramatos, J. Karpovich, A. S. Grimshaw. **Resource management in legion.** Technical Report CS-98-09, Department of Computer Science, University of Virginia, February 1998.

[8] **MPICH-G2**, http://www3.niu.edu/mpi/

[9] **Globus**, http://www.globus.org/

[10] K. Czajkowski, S. Fitzgerald, I. Foster, C. Kesselman. **Grid Information Services for Distributed Resource Sharing.** Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10), IEEE Press, August 2001.

[11] R. Wolski, N. Spring, J. Hayes. **The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing.** Journal of

Future Generation Computing Systems, Volume 15, Numbers 5-6, pp. 757-768, October, 1999.

[12] J. M. Orduna, V. Arnau, A. Ruiz, R. Valero, J. Duato. **On the design of communication-aware task scheduling strategies for heterogeneous systems.** Proceedings of International Conference on Parallel Processing, 2000. pp. 391 -398, 2000.

[13] J. M. Orduna, V. Arnau, J. Duato. **Characterization of communications between processes in message-passing applications.** Proceedings of IEEE International conference on Cluster Computing, pp. 91-98, 2000.

[14] C. Lu, D. A. Reed. **Compact Application Signatures for parallel and Distributed Scientific Codes**. Proceedings of SC'2002, Baltimore, MD, November 2002.

[15] R. L. Ribler, J. S. Vetter, H. Simitci, D. A. Reed. **Autopilot: Adaptive Control of Distributed Applications**. Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing.

[16] **Understanding X.500 – The Directory**, http://www.isi.salford.ac.uk/staff/dwc/Version.Web/Contents.htm

[17] G. von Laszewski, I. Foster, J. Gawor, W. Smith, S. Tuecke. **CoG Kits: A Bridge between Commodity Distributed Computing and High-Performance Grids.** In ACM Java Grande 2000 Conference, pages 97–106, San Francisco, CA, 3-5 June 2000.

[18] F. Berman, R. Wolski, S. Figueira, J. Schopf, G. Shao. **Application-Level Scheduling on Distributed Heterogeneous Networks.** Proceedings of Supercomputing 1996.

[19] **The Globus Resource Specification Language RSL v1.0**, http://www-fp.globus.org/gram/rsl_spec1.html

[20] C.W. Yeh, C.K. Cheng, T.T.Y Lin. **Circuit Clustering Using a Stochastic Flow Injection Method.** IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 14, No 2, February 1995.

[21] C.W. Yeh, C.K. Cheng, T.T.Y Lin. **A probabilistic multicommodity-flow solution to circuit clustering problems.** Proceedings of the 1992 IEEE/ACM International Conference on Computer-Aided Design, 1992.

[22] K. Taura, A. Chien. **A Heuristic Algorithm for Mapping Communicating Tasks on Heterogeneous Resources.** Proceedings of the Heterogeneous Computing Workshop 2000.

[23] J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra. **Using PAPI for hardware performance monitoring on Linux systems.** Presented at Linux Clusters: The HPC Revolution July, 2001.

[24] L. DeRose, D. A. Reed. **SvPablo: A Multi-Language Architecture-Independent Performance Analysis System.** Proceedings of the International Conference on Parallel Processing (ICPP'99), Fukushima, Japan, September 1999.

[25] **UDDI Version 3.0, UDDI Spec Technical Committee Specification**, 19 July 2002, http://uddi.org/pubs/uddi-v3.00-published-20020719.pdf

[26] **The Apache Jakarta Project, Tomcat Server**, http://jakarta.apache.org/tomcat/

[27] **Apache Axis**, http://ws.apache.org/axis/

[28] **Simple Object Access Protocol (SOAP) 1.1**, W3C Note 08, May 2000, http://www.w3.org/TR/SOAP/

[29] L. A. Sanchis. **Multi-way network partitioning.** IEEE Transactions on Computing, Vol. C-38, no. 1, pp 62-81, Jan 1989.

[30] B. W. Kernighan, S. Lin. **An efficient heuristic procedure for partitioning graphs.** Bell Systems Technical Journal, Vol 49, pp 291-307, February 1970.

[31] **PBS Technical Overview**, http://www.openpbs.org/overview.html

[32] **MAUI scheduler**, http://supercluster.org/maui/

[33] V. S. Adve, R. Bagrodia, J. C. Browne, Deelman, E., A. Dube, E. Houstis, J. Rice, R. Sakellariou, D. Sundaram-Stukel, P. J. Teller, and M. K. Vernon. **POEMS: End-to-end Performance Design of Large Parallel Adaptive Computational Systems.** IEEE Transactions on Software Engineering, Special Section of invited papers from the WOSP '98 Workshop, Vol. 26, No. 11, Nov. 2000, pp. 1027-1048.

[34] T. Yang, A. Gerasoulis. **DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors.** IEEE Transactions on Parallel and Distributed Systems, vol. 5, no. 9, Sep. 1994, pp. 951-967.

[35] A. Radulescu, A.J.C. van Gemund. **Improving Processor Selection Complexity in List scheduling Algorithms.** Proceedings 12th International Conference on Control Systems Science, April 1999.

[36] R. Raman. **Matchmaking Frameworks for Distributed Resource Management.** PhD Dissertation, University of Wisconsin - Madison, 2001.

[37] R. Buyya, D. Abramson, J. Giddy. **Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid.** Proceedings of the HPC ASIA'2000, the 4th International Conference on High Performance Computing in Asia-Pacific Region, Beijing, China, IEEE Computer Society Press, USA, 2000.

[38] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, S. Tuecke. **A Resource Management Architecture for Metacomputing Systems.** Proceedings of IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing, pg. 62-82, 1998.

[39] **MDS 2.2 User's Guide**, http://www.globus.org/mds/mdsusersguide.pdf

[40] **Grid Economic Services Architecture**, http://www.gridforum.org/3_SRM/gesa.htm

[41] **LSF - Load Sharing Facility Documentation**, http://wwwpdp.web.cern.ch/wwwpdp/bis/services/lsf/

[42] H. Dail, H. Casanova, F. Berman. **A Decoupled Scheduling Approach for the GrADS Environment.** Proceedings of SC 2002, Baltimore, November 2002.

[43] **MPICH**, http://www-unix.mcs.anl.gov/mpi/mpich/

[44] J. Novotny. **The Grid Portal Development Kit**. Concurrency - Practice and Experience, pp.1-7,2000.

[45] P. Bora, C. Ribbens, S. Prabhakar, G. Swaminathan, M. Chinnusamy, A. Jeyakumar, B. Diaz-Acosta. **Issues in Runtime Scientific Algorithm Selection for Grid Environments.** Challenges of Large Applications in Distributed Environments, HPDC-12, June 2003. (To appear)

[46] I. Foster, C. Kesselman, **The Grid: Blueprint for a New Computing Infrastructure**. Morgan Kaufmann Publishers, Orlando, FL, 1999.

[47] J. R. Rice, et al., **Solving Elliptic Problems Using ELLPACK**. Springer Verlag, 1984.

[48] **Working with LoadLeveler**, http://beige.ucs.indiana.edu/B673/node88.html

[49] **The ScaLAPACK Project**, http://www.netlib.org/scalapack/

[50] I. Foster, C. Kesselman, **The Grid: Blueprint for a New Computing Infrastructure.** Morgan Kaufmann Publishers, Chapter 12, pp. 279 – 310, Orlando, FL, 1999.

[51] I. Foster, C. Kesselman, **The Grid: Blueprint for a New Computing Infrastructure**. Morgan Kaufmann Publishers, Chapters 11 – 13, pp. 257 – 338, Orlando, FL, 1999.

[52] J. B. Weissman, A. S. Grimshaw. **A Federated Model for Scheduling in Wide-Area Systems**. Proceedings of 5th IEEE International Symposium on High Performance Distributed Computing, pp. 542-550, 6-9 Aug. 1996.

[53] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, D. Zagorodnov. **Adaptive computing on the grid using AppLeS**. IEEE Trans. Parallel and Distributed Systems, Vol. 14, No. 4, pp. 369-382, 2003.

Vita

Sandeep Prabhakar

Education

- M.S., Computer Science and Applications, June 2003 (GPA: 3.87/4.0)
  Virginia Polytechnic Institute and State University (Virginia Tech), Blacksburg, VA.
  Thesis: Zone Based Scheduling: A Framework for Scalable Scheduling of SPMD parallel programs on the Grid.

- B.S., Computer Science and Engineering, May 2001 (GPA: 8.5/10.0)
  College of Engineering Guindy, Anna University, Chennai, India.
  Thesis: Performance Improvement for Controlled Load RSVP Protocol for Delay-Sensitive Loss-Tolerant Multicast Applications.

Selected Projects

- Designed and Implemented a causally ordered peer-to-peer instant messenger using Java Spaces
- Developed a Tool for Financial Visualization using Swing API and JDBC
- Designed a LAN/WAN solution for a (hypothetical) startup dotcom company
- Designed an E-Commerce site using Servlets, JSP and JavaScript
- Developed a protocol for delay-sensitive, loss tolerant Multicast Applications using RSVP
- Parallel Error Correction Codes for the DLX processor (instruction level parallelism)
- Implemented a compiler for subset of C in lex and yacc
- Implemented a new Distributed Object system that handles remote objects
- Implemented a Meta Branch Predictor in the Simplescalar simulator
- Wrote a simulator for alpha 21264 Processor

Work Experience

- Graduate Teaching Assistant for Formal Languages: Department of Computer Science, Virginia Tech, Jan-May '03

- Graduate Teaching Assistant for Graduate Operating Systems: Department of Computer Science, Virginia Tech, Aug-Dec '02

- Instructor for Introduction to programming in C++: Department of Computer Science, Virginia Tech, Jun-Jul '02

- Graduate Teaching Assistant for Data structures and Algorithm Analysis: Department of Computer Science, Virginia Tech, Aug '01-May '02

Publications

- P. Bora, C. J. Ribbens, S. Prabhakar, G. Swaminathan, M. Chinnusamy, A. Jeyakumar and B. Diaz-Acosta, "Issues in Runtime Algorithm Selection for Grid Environments," in *Challenges of Large Applications in Distributed Environments: Proceedings of the International Workshop on Heterogeneous and Adaptive Computation*, June 2003.

- C. J. Ribbens, P. Bora, M. Di Ventra, J. Hauck, S. Prabhakar C. Taylor, and M. Di Ventra, ``From cluster to Grid: a case study in scaling-up a molecular electronics simulation code," in *Proceedings of the High Performance Computing Symposium, HPC2003* I. Banicescu (ed.), Society for Modeling and Simulation International, San Diego, pp 54-62, 2003.

- S. Prabhakar, C. Ribbens and P. Bora, ``Multifaceted web services: an approach to secure and scalable grid scheduling," in *The Web and the GRID: from e-science to e-business, Proceedings of Euroweb 2002*, B. Matthews, B. Hopgood, and M. Wilson (eds.), British Computer Society, Swindon, UK, pp 116--125, 2002.

- "Multiple Foci Drill-Down through Tuple and Attribute Aggregation Polyarchies in Tabular Data", *Proceedings of IEEE InfoVis 2002 Symposium*, October 2002.

- "Breakdown Visualization – Multiple Foci Polyarchies of values and attributes", *ACM – CHI*, April 2002.

- "Performance Improvement for the Controlled Load RSVP Protocol", *IASTED symposium*, Feb 2001.

- "An Architecture for Agent Based Intelligent Internet", *Applied Telecommunications Symposium*, April 2000.

Other Qualifications

- Languages: C, C++,VC++, Java, Assembly(x86), Prolog

- Operating Systems: Windows NT/ 98/ 2000/ XP, Linux, UNIX

- Technologies/Tools: MPICH-G2, Globus, TCP/IP, CORBA, SOAP, lex, yacc, SQL, ns2, OPNET, JNS, Simplescalar, Javaspaces, XML and web technologies

Courses of Interest

Graduate: Grid Computing, Operating Systems, Computer Network Architecture, Information Visualization, Internet Software, Theory of Algorithms, Network performance modeling.

Undergraduate: Software Engineering, Mobile Computing, Distributed Computing, Microprocessors, Compiler Design, Advanced Data Structures, Java Programming, Parallel Computing, TCP/IP.