Parallel and Sequential Job Scheduling in Heterogeneous Clusters: A Simulation Study using Software in the Loop

Dave E. Collins* and Alan D. George**

High-performance Computing and Simulation (HCS) Research Laboratory
*ECE Department, FAMU-FSU College of Engineering, Tallahassee, FL
**ECE Department, University of Florida, Gainesville, FL

The task of designing and optimizing job scheduling algorithms for heterogeneous computing environments requires the ability to predict scheduling performance. The complexity of heterogeneous scheduling issues requires that the experiments devised to test a scheduling paradigm be both flexible and extensive. Experimentally determined models for the prediction of job execution times on both sequential and parallel computing resources are combined with the implementation of a novel scheduling algorithm and a software-in-the-loop (SWIL) simulation. The result is a potent design and analysis approach for job scheduling algorithms and implementations intended for heterogeneous environments. This paper develops the concepts, mechanisms, and results of a SWIL design and analysis approach. The merits of this approach are shown in four case studies, which determine overhead, scheduling performance, and the impact of preemption and priority policies. These case studies illustrate the contributions of this research in the form of a new parallel job scheduling algorithm for heterogeneous computing and in the novel application of SWIL simulation to the analysis of job scheduling systems.

Keywords: Resource management systems, simulation, job scheduling, preemptive scheduling, priority scheduling, software in the loop, heterogeneous computing, cluster computing

1. Introduction

Commercial off-the-shelf (COTS) workstation clusters are probably the most powerful yet poorly employed computing resource available to most organizations today. Furthermore, many of these clusters are heterogeneous in architecture, operating system, or network. Often, this heterogeneity is not even planned, but arises simply due to the march of technology over time and the whim of the computer market. Heterogeneous computing research has shown that with careful job scheduling, heterogeneous collections of computing resources can usually outperform comparable homogeneous resource sets when the application set places varied demands on the computing nodes and the interconnection networks [11]. Considering the fact that workstation clusters have excellent price to performance ratios, and given that COTS heterogeneous clusters can realize a very high level of aggregate performance, why is it that heterogeneous clusters are rarely effectively applied outside of research efforts?

Part of the answer lies in inertia, since large organizations are often slow to adapt new technologies and computing paradigms. However, a more significant factor is that heterogeneous clusters are difficult to use effectively. The programmer and user generally need to be painfully aware of the fact that the underlying resources and networks are complex and varied, and must condition their actions based on this heterogeneity of resources to realize high performance in their applications. One possible solution is to develop a Single System Image (SSI) to hide the complexity of the heterogeneous resource set from at least the programmer and end user, while easing the burden on the system and network administrator. Older implementations of an SSI include IBM's *Parallel Sysplex* and the DEC/Compaq *VAXcluster* environments. More recent developments in the SSI area include the IBM *Phoenix, Microsoft Cluster Service* (*"Wolfpack"*), Berkeley's *GLUnix*, and Sun's *Solaris MC* [25].

An effective Resource Management System (RMS) is an important component of an SSI intended to improve the efficiency with which tasks are scheduled to computing resources. *Condor*, *Codine*, *NQS*, *NQE*, *Spawn*, *SmartNet*, *Loadleveler*, and *DQS* are a few examples of the currently available RMS implementations. With the exception of *SmartNet*, the tools described above all operate under the principle of OLB (Opportunistic Load Balancing) [12]. OLB essentially assigns one task to each machine, and gives each machine another task whenever one has been completed. However, *SmartNet* is significantly different, and uses the concept of an experiential database (i.e. a database holding information regarding the execution time distribution of the job's previous executions used to predict execution time) and a number of scheduling algorithms. An overview of *SmartNet* and job scheduling will be forthcoming, as the case studies analyze and evaluate a parallel job scheduling approach that is a novel evolution of the *SmartNet* concept.

The remainder of the paper is structured as follows. In the next section, an overview of related work in the area of job scheduling is presented and the novel parallel job scheduling system is placed into the context of the existing literature. In Section 3, an overview of the novel Parallel Job Scheduler (PJS) algorithm and its associated site database is given. In Section 4, the method is discussed for the employment of software-in-the-loop (SWIL) simulation around the existing PJS software. Section 5 includes the description, results, and analysis of four case studies conducted using this approach as well as the motivation in each case for choosing a SWIL approach. These case studies demonstrate how SWIL simulation can be used to investigate questions related to job scheduling algorithms, implementations, and policies. These results show that the PJS is capable of achieving up to twice the level of performance of its sequential ancestor and that it is particularly well suited for priority and preemptive priority scheduling. In addition, the results also determine the overhead of the PJS within the context of batch job scheduling. Finally, conclusions about the uses and contribution of the SWIL approach as applied to the analysis and design of job scheduling environments and RMS policies as well as future directions for research are discussed in Section 6.

2. Related Work

The following section describes related work in the literature. SWIL literature is first treated briefly and a review of literature related to job scheduling in heterogeneous computing environments is presented.

2.1. SWIL Methodology and Application

The ability to execute real software on simulated hardware has long been a desire for researchers and designers in the field of computing. In recent years, the means to achieve this goal have emerged. By running sequential program code on VHDL representations of processors, the co-simulation movement (also called co-design or co-verification) took the lead in attempts to combine application software and simulated hardware. Co-simulation is the process by which real application code, written in a high-level programming language such as C, is fed to a processor model, written in a low-level hardware description language such as *VHDL* or *Verilog* [1]. Contemporary to the co-simulation work, methods were developed to run applications on physical prototypes using reconfigurable FPGA hardware. In recent years, a new emphasis has emerged for parallel systems and their simulation. Several research groups have created methods for running parallel or multithreaded programs over simulated sharedmemory multiprocessors. None have as of yet proposed an approach for the use of cosimulation methodology for the development, evaluation, and refinement of job scheduling algorithms for heterogeneous environments.

Software-in-the-Loop (SWIL) simulation is an effective methodology for analyzing the functionality and performance of a software implementation. It differs from Hardware-in-the-Loop (HWIL) simulation in that the software used is the actual implementation and the hardware and testing environment is partly or wholly simulated, whereas in HWIL simulation the hardware is entirely real and the other elements may be thus simulated. SWIL simulation is a common capability in hardware/software codesign packages such as *Ptolemy*, developed at the University of California at Berkeley [6]. Such codesign environments require the ability to use a simulation of the hardware to test the actual software and vice-versa to realize their full potential in accelerating the design cycle. *ISE*, developed at the University of Florida, also provides SWIL capabilities for software testing, refinement, and performance evaluation in a parallel computing environment [13]. Another area where SWIL methodology is sometimes used is in the development of control software. *ATTSIM* is one example of a tool for SWIL simulation in this area, intended for the design and verification of flight control concepts, architectures, and algorithms [18].

This research describes the use and proves the utility of SWIL simulation techniques for the evaluation and analysis of implementations of job scheduling algorithms for heterogeneous computing environments. This approach represents a novel application of the SWIL simulation paradigm.

2.2. Job Scheduling in a Heterogeneous Computing Environment

Ideally, heterogeneous applications should be scheduled in an intelligent fashion to make best use of the available resources. Best use of the available resources, in this case, refers to a metric of performance (e.g. total completion time, average response time, or average throughput) chosen and weighted to reflect the values and priorities of the users [5]. One goal of this research is to develop a unified approach to parallel and sequential job scheduling for heterogeneous clusters. This approach is demonstrated by the development of the PJS, a novel scheduling algorithm. The discussion that follows places the PJS within the context of the existing literature.

There exists a number of job schedulers that attempt to efficiently match jobs to resources in a networked environment. Some examples include the *DQS*, *NQE*, *Condor*, *LoadLeveler*, and *SmartNet* RMS tools. There also exists a number of distributed operating systems upon which the parallelization of jobs is presumed to be the default, such as *Amoeba*, *Chorus*, *Mach*, *Sumo*, and *Spring* [17]. Such distributed operating systems necessarily perform scheduling on incoming jobs, as they aim to present an SSI to the user. There also exist parallel coordination languages, such as *Piranha Linda*, which make some provision to the scheduling of resources and tasks [27]. The PJS described later in this research is best described as a scheduling algorithm well suited to a RMS.

OLB is by far the most prevalent algorithm for scheduling jobs to machines with a conventional RMS. In simple terms, OLB involves scheduling one task to each resource unit (e.g. a processor, cluster, or parallel machine). If there are jobs remaining, the machines first completing their jobs will be assigned another task until all jobs are complete. If there are fewer jobs than resource units, the resources first responding will receive jobs.

Best Assignment (BA) is also employed on occasion in an RMS. With best assignment, each task has a designated type of resource that it is best suited to using. The scheduler then attempts to assign that job to a resource unit matching that description. Sometimes this approach is combined with OLB in scheduling. This hybrid approach uses job requests that specify on which type of machines the user would prefer the job be executed, and the other, less preferred, machines that the job is capable of being executed on if necessary. This scheduling technique is often used with the *PVM* (Parallel Virtual Machine) coordination language [28].

Hagerup presents the Bold scheduling algorithm, which assumes that the mean and variance of execution times are known a priori [14]. It also assumes a fixed delay when jobs are dispatched. Bold attempts to minimize the makespan (i.e., the time necessary until the last task has completed). Bold adaptively decreases the batch size of jobs assigned as time progresses in the attempt to ensure that each batch assigned will not be the last to complete. Bold is not, however, designed to exploit the heterogeneous environment of *SmartNet* or the PJS.

With the exception of *SmartNet*, all RMS implementations at this point in time employ relatively simple scheduling algorithms (i.e., OLB and BA) to make their scheduling decisions. *SmartNet* thus marked a step forward in the evolution of RMS tools, as it supports the use of a large number of scheduling algorithms. There is one key assumption in *SmartNet's* design: that past executions of a job type contribute useful information in the prediction of future job execution times. The reader is referred to [21] for a more complete treatment of this assumption. Techniques for the prediction of parallel application performance yielding relatively modest worst-case errors are also explored in [24]. It is also implicit in the design of *SmartNet* that its users deal with a tractable number of job types with relatively predictable execution characteristics. This knowledge is necessary to improve scheduling performance over OLB, and the accuracy of this data greatly influences that performance [12].

It should also be mentioned that there is a good deal of research into the area of task scheduling. Generally, this scheduling research is concerned with the execution of a single job, which is broken into a number of tasks that may have precedence constraints and/or communication requirements. In such research, the focus is usually on reducing the makespan. Directed acyclic graphs are used to depict the subtasks of the job, where a circle is used for each subtask and an edge with a weight indicates the necessity and magnitude of any communication. Task clustering, where several subtasks are assigned to a single machine, is one method used to minimize the makespan, as the communication cost within a single machine is presumed to be negligible. In cases where the communication to computation ratio is large, it is sometimes profitable to use task duplication (i.e., where the identical task is performed locally on several machines to avoid the communication overhead needed to transmit its data). The reader is referred to [23] and [34] for a more extensive discussion of this topic. Other research into approaches for task scheduling can be found in [4,9,15-16,19,29-30].

These algorithms differ from the PJS algorithm developed in this research because they assume complete a priori knowledge of execution times rather than statistical knowledge (i.e., mean and variance) of execution times. They also differ from this research in that they generally deal with tasks that collectively form some larger job. This focus explains why such algorithms generally use the makespan as their figure of merit as the completion of the component tasks only becomes meaningful once the entire set is finished. The PJS focuses on the job, which may itself be a collection of tasks. For example, a job scheduler might assign a job that consists of a group of tasks to a cluster. That collection of tasks could then use a task-based scheduling algorithm to minimize its makespan given the resources allocated to it. Thus, there is a natural nexus between these two scheduling approaches.

An approach closely related to task scheduling, referred to as application-level scheduling, similarly attempts to optimize the performance of a single job, usually composed of multiple tasks. One implementation of application-level scheduling is the *AppLeS* (Application Level Scheduler), which is capable of using the related tool *NWS* (Network Weather Service) to obtain information about resource availability although, in contrast to task scheduling approaches, it makes no assumption of a priori knowledge of execution times [31-32].

3. PJS Algorithm and Database

Job scheduling is a key issue and a nontrivial problem when considering single-processor, stand-alone machines, and a number of scheduling algorithm options are available for their operating system (OS) design. For a heterogeneous cluster or group of clusters, the problem is far more complex. Yet, at the same time, considerable performance gains can be realized if the jobs assigned to the cluster or clusters can be scheduled in an optimal, or near-optimal fashion. While the general problem of scheduling has been proven to be an intractable one for the majority of cases, scheduling algorithms based on optimization mathematics and general heuristics have proven themselves to be useful [5]. The research effort described this section outlines a novel approach to generating a near-optimal solution to this problem in the presence of uncertainty.

SmartNet marks the current state of the art in terms of job-based (rather than task-based) scheduling for heterogeneous computing. However, it does have a number of limitations in its basic model, most notably the lack of an ability to represent a single resource as being available both as a computational unit in and of itself and as a member of a more powerful cluster. Research has been conducted into addressing some of the limitations of *SmartNet*. An interface was developed allowing *SmartNet* to make use of parallel applications written for the *Linda* coordination language. This interface was developed, tested, and verified in simulation in [8].

From this prior research, it is clear that scheduling performance can be improved substantially by allowing a job scheduler to decide where and when to execute jobs in parallel. By loosening some of restrictions in the current *SmartNet* model, while maintaining much of the same approach and principal assumptions, substantial performance improvements may be realized through the use of the PJS algorithm. The working hypothesis for job scheduling within a heterogeneous cluster is as follows:

Through the use of experiential data regarding both parallel and sequential computing resources, aggregate job performance can be improved using scheduling algorithms that impose modest overhead.

Within the context of this research, most advantageous job schedule is defined to be the schedule for that job queue that yields the smallest summation of expected execution and delay times. This metric is distinguished from the expected makespan, which is the amount of time expected to be necessary until the last job in the queue has completed. The novelty of this approach is that it represents a fusion of both parallel and sequential experiential data and supports the aggregation of computational resources in an intelligent manner. The PJS was developed to prove this hypothesis. An overview of the PJS approach is provided in the paragraphs to follow.

The PJS maintains a database for each site. A site is defined as a group of machines that may be used indifferently by a set of users. Thus, machines at a site will generally share substantial portions of their file systems and user databases, a feature of definite assistance for remote job execution. Within a site are machines, interconnects, clusters, users, job classes, and job queues. The PJS maintains information regarding the current and previous states of these objects. The database organization used by the PJS is depicted in Figure 1.

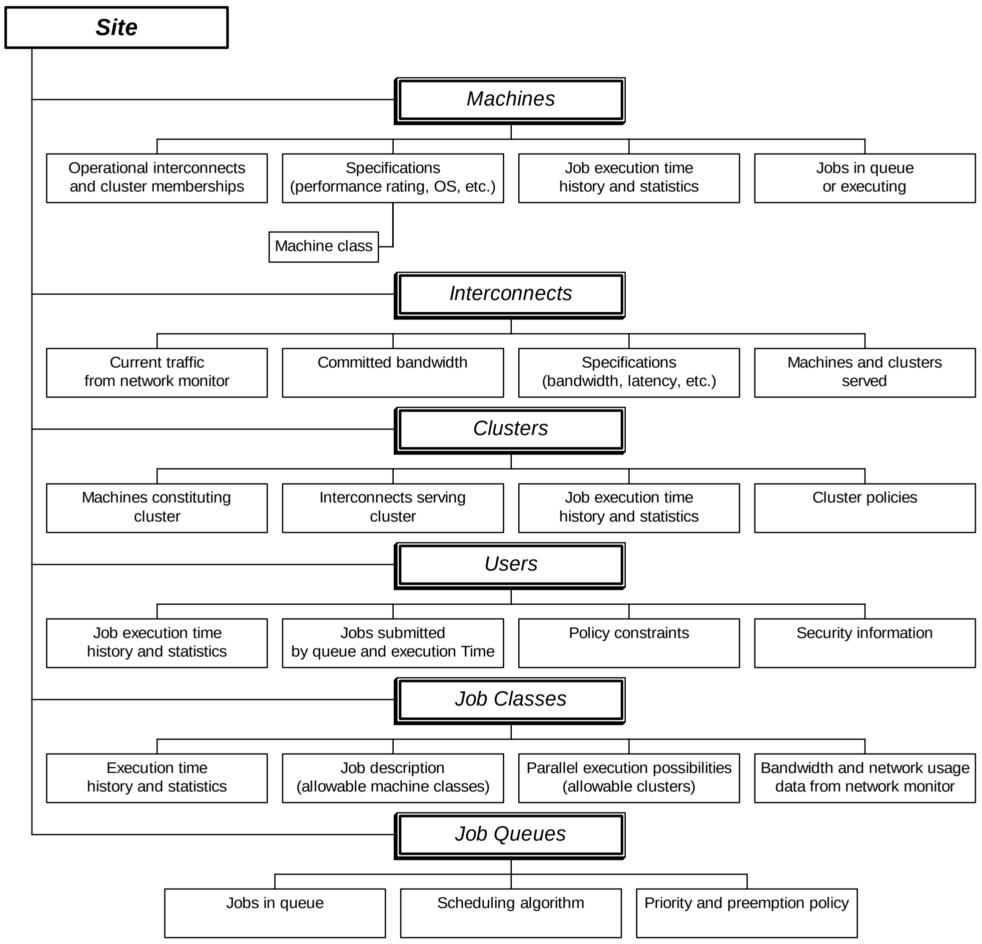
Site
Machines
Operational interconnects and cluster memberships
Specifications (performance rating, OS, etc.)
Job execution time history and statistics
Jobs in queue or executing
Machine class
Interconnects
Current traffic from network monitor
Committed bandwidth
Specifications (bandwidth, latency, etc.)
Machines and clusters served
Clusters
Machines constituting cluster
Interconnects serving cluster
Job execution time history and statistics
Cluster policies
Users
Job execution time history and statistics
Jobs submitted by queue and execution Time
Policy constraints
Security information
Job Classes
Execution time history and statistics
Job description (allowable machine classes)
Parallel execution possibilities (allowable clusters)
Bandwidth and network usage data from network monitor
Job Queues
Jobs in queue
Scheduling algorithm
Priority and preemption policy

Figure 1. Database organization for the PJS.

The first component of a site is its machines. Each machine has a number of resources, as illustrated in Figure 1, whose state is ideally known by the job scheduler. While a job scheduler will obviously be aware of all the jobs it has scheduled to a machine, it may not be aware of the interactive tasks running on that machine. This information can be provided by a resource-monitoring tool such as PARMON [7]. System specifications, such as number of processors, memory, stable storage space, and operating system must also be known, to permit the user and the job scheduler to know if a job can actually be executed on that machine. For example, a binary will not typically run without recompilation on a machine with a different OS from its original target. Similarly, certain jobs require vast amounts of physical memory to execute in a reasonable amount of time. These resource requirements are abstracted into the notion of a machine class. The level of abstraction of a machine class is left as a policy decision to the site's support staff. At a minimum, a class should indicate that all executables that typically run on one member of a class can be run on all members of a class. Furthermore, the operational interconnects servicing the machine and those clusters that the machine is a member of must be known. The state of these interconnects can be provided to the scheduler by a resource monitoring program. Lastly, the jobs currently executing and the job execution time history for each machine and cluster are maintained in the experiential database. This information allows the job scheduler to make an informed estimate of the execution time of each new job.

The next part of a site is its interconnects. Each interconnect is defined in terms of its specifications such as bandwidth and latency and which machines and clusters are connected by it.

Another important piece of information for a job scheduler is the clusters available. Each cluster is defined as a group of machines capable of executing a job in parallel. For example, a group of six Sun Ultra 2 workstations connected via a high-speed data network might be designated as a cluster. The cluster also is defined by those interconnects that serve it. Since each machine within a cluster is also a candidate for sequential execution, it is also important to define cluster policies. Specifically, at what level of available cluster membership should parallel execution be rejected? A clear statement of this policy is particularly important when the cluster membership grows into the tens and hundreds, as it is probable that at least one machine will be unavailable.

Another important component of a site is the users. The number of jobs submitted by each user and the execution times of said jobs is retained in the experiential database of the PJS. The particular job queue that the job is submitted to is recorded in the experiential database. The policies set by the site's support staff governing each user will also be recorded in the experiential database. Finally, provision for retaining security information for each user is made in the experiential database.

Each site also maintains a set of job classes. These classes are an abstraction that may be set to the desired level of specificity by the administrative staff at the site. A job class represents a set of potential jobs that have statistically correlated execution times when executed on a given parallel or sequential resource. This correlation is the justification for aggregating the execution time data for each (job class, computing resource) pair. Execution time prediction from this data uses the same procedure as in *SmartNet* [12].

Lastly, the site maintains a set of job queues. Each job queue maintains its own policies regarding preemption, priority, and user access. Furthermore, each job queue maintains a record of each job submitted to it. Also, each job queue will have a scheduling algorithm associated with it. The scheduling algorithm assigns the jobs in the queue to the appropriate parallel and sequential resources. These algorithms will be detailed later in this section.

An overall goal for the PJS is the minimization of the overhead of its software implementation. Due to this objective, the level of centralization has been decided to be a single multithreaded daemon per schedulable machine. This worker daemon accepts jobs sent to it by the manager, executes them, and returns experiential information to the manager. One overall multithreaded manager daemon maintains the complete experiential database and communicates with the client and any other tools in use.

The job queues have several scheduling algorithms available to them. The scheduling algorithms implemented are as follows:

1. The OLB algorithm.
2. The duplex greedy algorithm (best of the min-min and min-max scheduling algorithms), hereafter referred to as the Sequential Job Scheduler (SJS) algorithm and described later in this section.
3. The duplex greedy algorithm, with parallel cluster scheduling extensions, hereafter referred to as the PJS algorithm and described later in this section.

OLB was selected for implementation in the PJS primarily for comparison purposes for future research, since OLB is the primary scheduler in most RMS tools. OLB, due to its extreme simplicity, is also not computationally demanding, which indicates that it may be a good choice for queues where extremely large numbers of relatively short jobs are scheduled.

The second algorithm implemented within the job scheduler is the SJS algorithm. This algorithm takes the min-min and min-max scheduling algorithms and chooses the result which yields the most advantageous schedule.

The min-min algorithm first determines the best resource for each job in the schedule and the amount of time necessary for its completion on that resource. Best, in this context, refers to that resource that has the smallest expected execution time for that job. Then the job with the smallest amount of time required for its completion is scheduled on its best resource and the time of next availability is updated for that resource. This process is then iterated until all of the jobs have been assigned to resources. Min-max uses the same procedure, but it schedules the job with the largest, rather than the smallest, amount of time required for completion at each step. The SJS algorithm is the primary algorithm used by *SmartNet*, and is included within the PJS implementation for purposes of comparison.

The final scheduling algorithm implemented is a novel extended version of the SJS algorithm. This algorithm consists of the following steps:

1. Use the SJS algorithm in the same manner as in *SmartNet* to develop a job schedule where all jobs are executed sequentially. This schedule will form a baseline for improvement. Compute also the expected summation of execution times for this schedule.
2. Construct a list of all sequential jobs that may be executed in parallel.
3. For each possible parallel (job, cluster) pair, compute the reduction in expected execution time from the best sequential execution option for that job. This number is hereafter referred to as the profitability.
4. Allocate the necessary resources for parallel execution to the most profitable remaining parallel job. Commit the resources used by this job for the expected duration. If there are no profitable jobs then stop and use the best schedule determined thus far.
5. Using the SJS algorithm, schedule the sequential jobs around the jobs assigned to clusters thus far and compute the new expected summation of execution times. If the parallel job improved the schedule according to this metric, repeat steps 2-5 until the resulting schedule fails to be an improvement. At this point, use the best schedule determined thus far.

It should be noted that this algorithm always produces a result that is superior or equal to the SJS algorithm. Therefore, its costs over that algorithm lie solely in the overhead of its implementation.

4. SWIL Simulation Framework

SWIL simulation is one of the only viable approaches to the analysis of a job scheduling system without both a complete prototype and access to all of the resource set to be used. Using this approach, one can develop implementations of only the scheduling algorithms and the basic interface to the site database rather than a complete RMS. With only the "brain" of the scheduling system implemented in actual code, a SWIL framework is then constructed, providing a very powerful and flexible platform for analysis and refinement of the proposed system.

Four case studies have been identified to investigate the performance, overhead, and capabilities of the PJS and the SJS approaches using SWIL methodology. These case studies include the determination of the overhead imposed by the implementations of the PJS and the SJS algorithms, the comparison of performance of the PJS and the SJS algorithms over a given job set and job load, the impact of priority scheduling on performance absent preemption, and the effect of preemptive priority scheduling on performance. Each of these case studies is enhanced through the use of SWIL simulation, as will be demonstrated in Section 5.

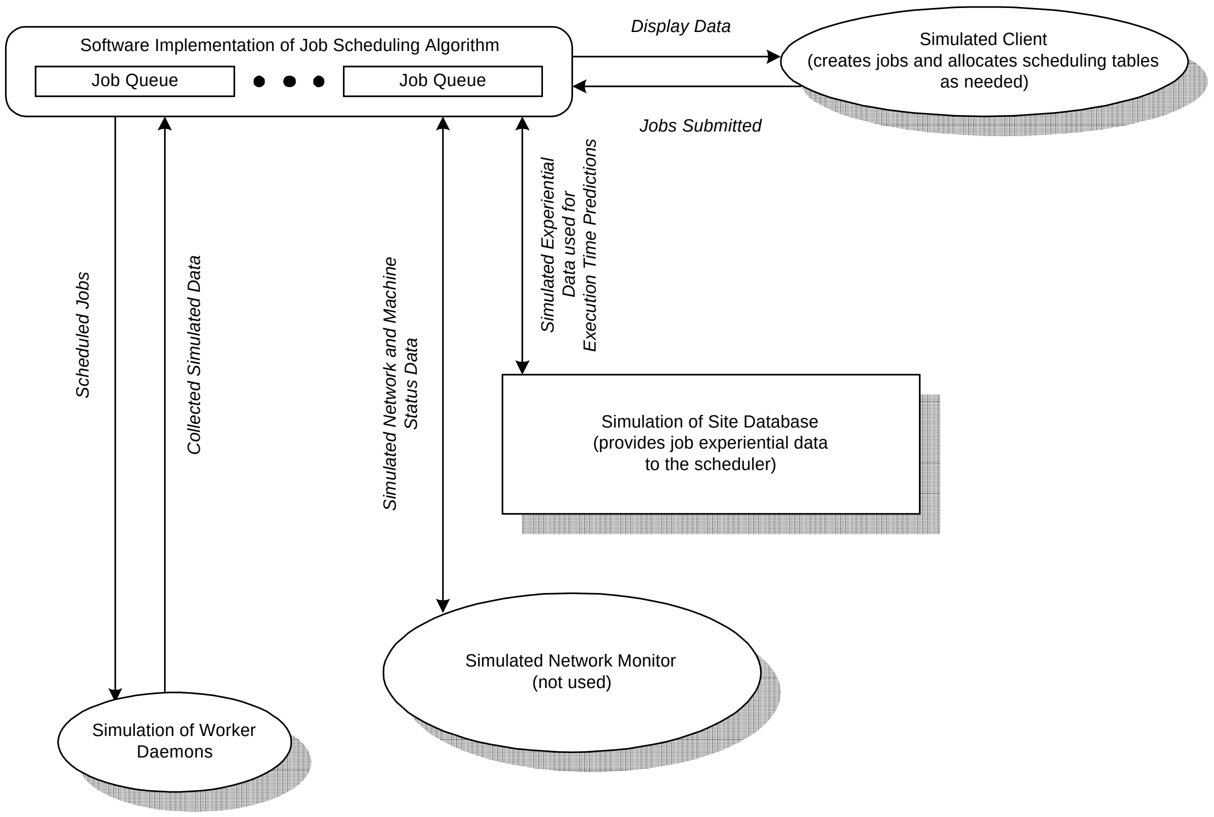
Software Implementation of Job Scheduling Algorithm
Job Queue
Job Queue
Display Data
Jobs Submitted
Simulated Client (creates jobs and allocates scheduling tables as needed)
Scheduled Jobs
Collected Simulated Data
Simulated Network and Machine Status Data
Simulated Experiential Data used for Execution Time Predictions
Simulation of Site Database (provides job experiential data to the scheduler)
Simulated Network Monitor (not used)
Simulation of Worker Daemons

Figure 2. Configuration of the Software-in-the-Loop (SWIL) approach to job scheduling simulation.

Figure 2 depicts the configuration for the SWIL simulation approach used for the evaluation and analysis of the PJS and the SJS software implementations. All of the simulation models were constructed using object-oriented

C++ code. The simulated client creates the jobs and allocates scheduling tables of the appropriate size. These scheduling tables are actually processed by the software implementation of the PJS or the SJS algorithm that is "in the loop." The simulation environment also provides the experiential data necessary for scheduling predictions. This experiential data was experimentally determined and thus validated by performing a series of test runs of each of the applications on the machines and clusters within the resource set. The SWIL environment used also contains provision for a simulated network and machine status monitor. Although not used in any of the four case studies, this provision may find use in future research. Finally, the simulation environment simulates the actual completion of the jobs according to the experimentally determined sequential or parallel execution times. This data is provided to the job scheduling software, as it is important for the determination of when resources are next available. This simulation of job execution also interacts with the preemption policy in the fourth case study, as a simulated job needs to be halted and rescheduled if a simulated preemption occurs. A fairly similar arrangement for the simulation of various job scheduling algorithms is used in [2], although all components in that arrangement including the scheduling algorithms are simulated and all the algorithms considered are for scheduling only sequential jobs.

5. Case Studies

A description of the four case studies and the experiments devised to investigate them is provided in this section. Each case also includes a discussion of the results and analysis for that case study, as well as the factors that motivated the use of the SWIL method.

5.1. Case Study 1: Scheduling Overhead Experiment with SWIL Simulation

The first experiment run on the PJS was devised to determine and compare the overhead for both the implementation of the SJS algorithm and that of the PJS algorithm. The determination of the overhead of the SJS algorithm is carried out with the number of machines and the number of jobs as variables ranging from 5 to 100. Because the number of machines and the number of jobs was allowed to vary over a broad range, the method of simulation was chosen. A strictly simulative approach could be used here, but a SWIL simulation allows more faithful measurement of the overhead of a particular implementation of an algorithm running on a particular machine. The job and machine sets were input to the actual scheduler objects and the schedule generated was then discarded, as the machine set was purely simulated. The machine used for the overhead experiments was a 200Mhz UltraSPARC workstation with 128 MB of main memory running *Solaris V2.5*. The overhead of the scheduler for a particular number of machines and number of jobs data point was determined by subtracting the timestamp of the starting time for that particular iteration from the ending time of said iteration. The high-resolution system clock was employed. For the scheduling table itself, random values were filled in for the estimated time of completion for each (job, machine) pair. This assignment was made because for the sequential algorithm, the actual contents of the scheduling table do not affect the time needed to process it into a final schedule. This statement is not true in general for the PJS trials, where the suitability of the job set for parallel execution may drive the overhead.

Figure 3 shows the overhead of the SJS as a function of the number of jobs and the number of machines. In this figure, it is observed that the overhead grows with both the number of machines and the number of jobs in the particular iteration. According to theory, the execution time trend should eventually approach $O(n^2m)$, where n is the number of jobs and *m* is the number of machines [2]. This chart displays a relationship more akin to *O(nm)*, which is consistent with the explanation that the process of memory management (e.g., scheduling tables must be allocated and deallocated and have a size proportional to *nm*) is the dominant factor for the values of *n* and *m* investigated. The most important insight here is that the scheduling overhead is at all times very modest (generally well below 0.2 seconds), particularly when compared to the expected run times of typical batch jobs. There is one unusual peak in the center of the chart caused by the need to acquire additional memory from the operating system.

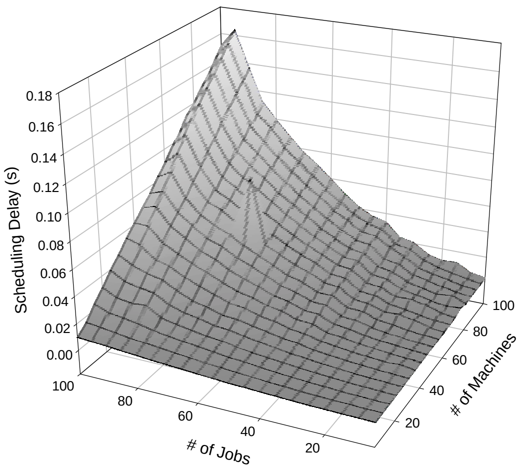

Figure 3. Scheduling delay of the SJS.

Next, the overhead of the PJS was determined. Here a 40-machine resource set was stipulated, which contained five clusters of four machines each and twenty "loose" machines. Recall that the PJS recognizes the dual nature of the machines within a cluster. Each is scheduled both as an individual machine and as part of a cluster. When scheduled a job as an individual machine, that machine's availability is pushed outward by the estimated time of completion. This modification affects the time at which the cluster itself becomes available, as the cluster is available only when all its components are ready. Scheduling a job to a cluster moves the available time of all its components outwards as well as its own available time. Several cases were considered: no possibly parallel jobs, 10% possibly parallel jobs, 50% possibly parallel jobs, 75% possibly parallel jobs, 90% possibly parallel jobs, and 100% possibly parallel jobs. A job is possibly parallel if there is at least one cluster on which it can run in parallel. It is possible that many jobs will not have a parallel implementation available. For instance, the speedups expected may not justify implementing an application for parallel execution. Once again, the execution times were assigned randomly for sequential (job, machine) pairs. The parallel cases were assigned randomly as well, but with approximately one-third the mean estimated time of completion. This assignment was made so that the scheduler would have a reasonable probability of finding favorable parallel options. The number of jobs was allowed to vary from 5 to 1000, and overhead determination used the same SWIL technique as in the previous experiment.

Figure 4 depicts the average of the scheduling delays for each of the cases considered over a range of 5 to 1000 jobs. The PJS typically takes from two to three times longer to execute than does the SJS (i.e., the 0% possibly parallel case).

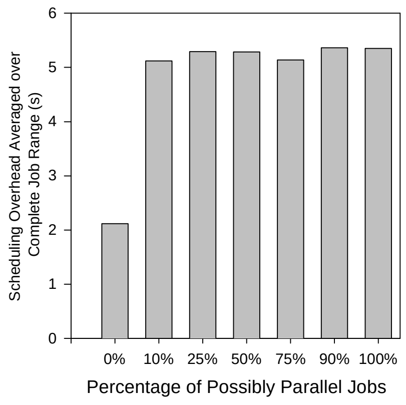

Figure 4. Average PJS overhead compared by case.

Longer execution times with the PJS are expected, since additional steps are required with this algorithm. The PJS will first compute a sequential job schedule. Next, it will find the most profitable job for parallel execution. If there are no profitable jobs, the sequential job schedule is used. Otherwise, the most profitable job is then scheduled in parallel and a new sequential job schedule is computed around it. This process is continued as long as the schedule's performance metric improves. When the addition of another parallel job to the schedule fails to improve it, the best schedule thus far is used and the process terminates. Thus, we expect the execution time to increase by roughly the amount of time needed for a sequential schedule for every parallel job that is fit into the final schedule. The averages in Figure 5 are consistent with that expectation. It is also noteworthy that the percentage of jobs that are candidates for parallel execution appears to have no significant overall effect on the amount of time needed to compute a schedule. This observation is explained by the fact that scheduling algorithms are designed to optimize overall performance over the entire set of jobs given them. Since parallel jobs rarely approach unity parallel utilization, only a few jobs will typically be scheduled in parallel under conditions of heavy load by the PJS. In a large job set under conditions of heavy load, having 10% of the jobs as candidates for parallelism is not much more restrictive in terms of the useful options for the PJS than having all of the jobs as candidates. Further, the amounts of time required to compute a schedule, even in cases where a very large number of jobs are considered simultaneously, are modest by the standards of batch jobs.

There are significant advantages to taking a simulative approach in this case study. First, the job set can be quickly input into the scheduler without need for the entry of information that is irrelevant to the particular purpose of the experiment. Job submissions normally include such things as command lines for each machine class or cluster that the user deems fit to execute the code in question. When the objective is simply to calculate the overhead imposed by the scheduler, the particular command line is unimportant, as the jobs need not be actually executed. The SWIL simulation modules can simply intercept the job dispatch orders that would normally be sent to the worker daemons. This technique allows the jobs to simply be scheduled, not executed, which is a serious consideration when simulating the scheduling of large numbers of jobs. By using this technique, it is feasible to perform a scheduling overhead analysis of a far larger job set than would be practical otherwise. Yet another benefit of this simulation technique is that any machine within the resource set may be real or simply notional. So long as the performance characterization data is made available to the scheduler, notional machines can be scheduled along with real machines. In this experiment the use of SWIL simulation allows the exploration of a considerably broader range of resource and job sets than would be feasible through conventional means while retaining the fidelity conferred through the use of the actual software implementation running on the actual target machine.

In summary, this case study shows that the overhead of the SJS is small over the range of *n* and *m* analyzed and the overhead of the PJS is from two to three times greater. The percentage of jobs that are candidates for parallel execution does not appear to significantly affect the overhead of the PJS, which indicates that the PJS algorithm is efficient in its search for parallelism.

5.2. Case Study: Scheduling Performance Experiment with SWIL Simulation

This experiment was performed to determine the performance improvement, if any, that would be realized by using the PJS algorithm versus the SJS algorithm. Only the PJS algorithm has the capability of scheduling jobs to run in parallel on clusters. Just how much of a difference this capability makes is what the experiment was devised to determine. The parameters of the experiment follow:

- The scheduler was tested using SWIL methodology (i.e., the schedulers were directly fed experiential data and used simulated dispatch facilities).
- The machine set that was used was constant and reflected the machines then present at the research site (142 machines, forming 18 clusters).
- Machines were generally grouped into clusters of 8 identical machines, with one cluster having only 6.
- The machines classes included several varieties from the Sun SPARC architecture family, and several varieties from the Intel Pentium architecture family. This machine set and cluster arrangement was used for this and subsequent case studies.
- The job set consisted of an equal mix of five job classes.
- Execution times for each (job, resource) combination were determined experimentally to produce the experiential data for the schedulers for both sequential and cluster resources. This job set and its experimentally determined execution times were used for this and subsequent case studies.

The five job classes for the test cases were chosen to reflect a reasonably diverse set of computing requirements. The first job class was a vector-matrix benchmark. This benchmark reflects floating-point performance and is typical of data parallel code. Parallel code was available for this benchmark as applications of this type are often parallelized. The second job class was a numerical integration benchmark. Floating-point performance is stressed, and parallel code was available for this benchmark as well. The third benchmark was an in-core integer sort. Parallel code was available, but speedups were modest. The fourth benchmark was a statistical analysis application. It used a mixture of floating-point and integer operations and was strictly sequential. The final benchmark used was the Numerical Aerospace Simulation (NAS) conjugate gradient benchmark. The conjugate gradient benchmark solves an unstructured sparse linear system using the conjugate gradient method and parallel code was available [3].

Both parallel and sequential performance for these benchmarks varied considerably over the resource set. The job set was not chosen to favor any particular type of machine, but there was considerable heterogeneity to be exploited.

Figure 5 shows that the performance of the PJS is in all cases at least equal to that of the SJS. We expect to see significant performance improvements under conditions of light load, as the cluster resources can then be employed without overly slighting the sequential jobs. Such behavior is in evidence, as Figure 5(a) shows percentage savings of total job time that range up to nearly 50%. In addition, there are also small performance improvements that occur periodically under conditions of moderate and heavy load, although they are too small to be seen in this chart. Such modest improvements reflect the ability of the job scheduler to occasionally find a profitable parallel job even when the cluster is heavily loaded.

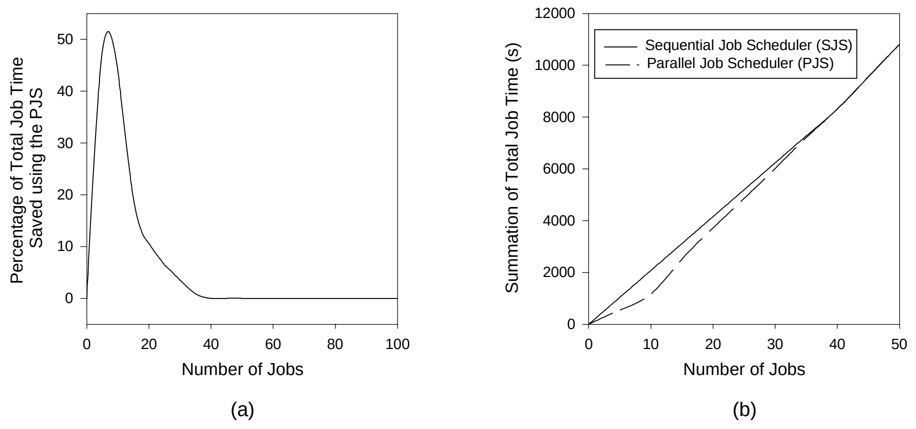

Figure 5. Performance improvement realized using the PJS (a), and performance comparison of the PJS and the SJS (b).

There are significant advantages to employing simulative techniques. In this case and in the case studies to follow, SWIL techniques were chosen over purely simulative methods because the SWIL framework constructed to faithfully analyze the first case study required very little modification to enable further case studies.

By running the application code that forms the job set on the various machine classes and clusters under controlled circumstances, the experiential data needed by the job scheduler can be determined without much of the error typically encountered. This error comes about because it is often difficult to obtain totally exclusive access to all of the machines in a site for an extended period of time. Thus, the interactive and other jobs that the normal users at the site are likely to perform in the course of their daily routine are likely to distort measurements of job performance on particular machines. This distortion would be less of a problem were there no particular bias for or against particular machines at the time of testing, but experience suggests that such is unlikely to be the case. By testing each particular (job, resource) combination in isolation, better control over outside interference and the error thus introduced is possible.

By intercepting the job dispatches as in the previous experiment, we are also freed from the necessity of actually running the scheduled jobs at the time when the experiment is conducted. Since batch jobs are typically quite large, and performance over a large range of load conditions is desired, actually executing the jobs in question could consume a prodigious amount of computing time and network bandwidth. Such consumption would be very likely to draw the ire of other users of the same resources, particularly if exclusive access was necessary. Another benefit that could be exploited is the use of notional machines along with the real machines in the resource set. This particular feature was not used in this experiment but it would be a powerful tool for deciding which, if any, new machine types to add to a resource set. By adding additional notional machines or clusters, the performance impact on a typical job set could be determined for each option and informed decisions about resource acquisition could then be made. Due to the heterogeneity in many COTS-based environments and their typical jobs, the most effective marginal addition to the resource set may not be obvious due to the gains possible from allowing additional "specialization" by machines in jobs for which they are particularly suited. Just as in the discipline of economics, possible gains due to specialization often require analysis before becoming apparent. The benefits of allowing a compressed form of the experiential data and the jobs to be scheduled to be input directly into the scheduling module are also again present. Changing the job set, resource set, or the experiential data for (job, resource) combinations is a simple matter when done in a simulation environment.

In summary, this experiment shows that the PJS has succeeded in its objective of making possible significant performance improvements over the SJS. At the same time, the PJS retains the performance of the SJS under conditions of heavy load.

5.3. Case Study 3: Priority Scheduling Experiment with SWIL Simulation

Priority scheduling is a feature often desired within a job scheduling system. It seems intuitively clear that the addition of a priority scheme will degrade the overall performance of the system to some degree (although, presumably the higher priority jobs will receive a performance benefit), however there is little literature available that quantifies this burden in the context of heterogeneous computing environments. Thus, an experiment was constructed to measure this cost when using the PJS. For purposes of comparison, the performance of a slightly modified SJS was also analyzed.

The priority system used has three priority levels: high, medium, and low. All the jobs were immediately presented to the job scheduler at time zero in the appropriate job queues. The job scheduler then completely processes the high-priority queue. Afterwards, the medium-priority and then the low-priority jobs are processed. Finally, all the jobs are dispatched, although this dispatch is intercepted by the simulation modules.

The number of jobs was allowed to vary up to 500 jobs, and four priority distributions were considered. The first priority distribution had 10% high-priority jobs, 10% medium-priority jobs, and 80% low-priority jobs (i.e. hereafter signified as 10/10/80). This distribution was intended to represent a typical job-scheduling environment, where a few jobs are high-priority but most are not. The second distribution had 25% high-priority, 25% medium-priority, and 50% low-priority jobs (i.e. hereafter signified as 25/25/50). This distribution reflects a site that makes heavy use of priority levels. The third distribution had 50% high-priority, 25% medium-priority, and 25% low-priority jobs (i.e. hereafter signified as 50/25/25). This distribution represents a site that sells its idle cycles but wants to insure the performance of its own jobs. That site would assign its own jobs a high or medium priority and relegate lent or sold "cycles" to a low priority. The fourth and final distribution had no priority levels and is included for purposes of comparison. Priority assignment becomes particularly important in the final case study, where the interaction of preemption and priority is investigated. In addition, the SJS was modified to use priority queues in the same fashion to allow for comparison.

Thus, this experiment investigates the performance impact of using priority levels in job scheduling. Both the sequential and parallel algorithms are compared, and four different priority distributions are considered. Furthermore, the job load seen by the scheduler is allowed to vary as well. While the resource set is held constant in this case study, the SWIL environment makes changing the resource set relatively straightforward, a feature that may find use in future research into priority scheduling.

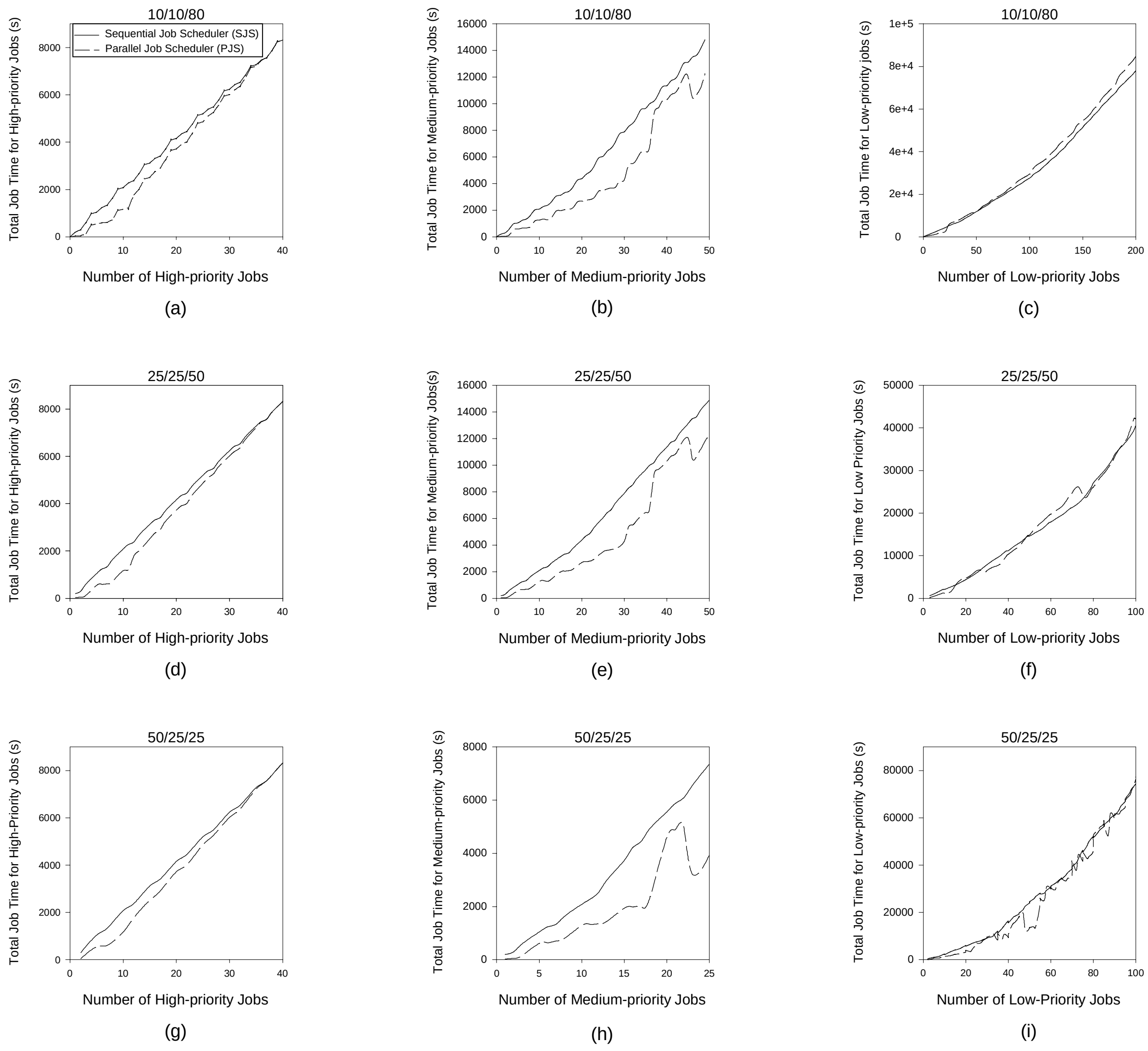

Figure 6. Comparison of PJS and SJS performance for priority scheduling for 10/10/80 priority distribution ((a), (b), (c)), for 25/25/50 priority distribution ((d), (e), (f)), and for 50/25/25 priority distribution ((g), (h), (i)).

The charts of Figure 6 were generated using the results of a series of SWIL simulations for each of the four job priority distributions. Each job priority level and job priority distribution is treated in a separate chart of the figure. Each point on a chart depicts the summation of the job times for those jobs at the indicated priority level as a function of the number of jobs at that priority level for the specified priority distribution. The total number of jobs (i.e., aggregating all priority levels) ranges from 5 to 500. It should be noted that jobs at a particular priority level coexist with jobs at the other priority levels as defined by the job priority distribution. For example, the point representing 5 medium-priority jobs in the 10/10/80 distribution represents the scheduling of 5 high-priority jobs and 40 low-priority jobs as well, although only the summation of the total job times for the 5 medium-priority jobs is displayed. Only the regions of interest are depicted in the figure and results are shown for both the PJS and the SJS.

Figure 6 shows that the PJS obtains performance superior to the SJS for high-priority jobs. This behavior is the objective, as the PJS can favor high-priority jobs more than the SJS by executing them in parallel and thereby obtaining speedups. The extent to which it can do so is directly related to the rarity of high-priority jobs. Figures 6(a), 6(d), and 6(g) support this observation. It can be concluded that the PJS algorithm gives greater ability to optimize the performance of high-priority jobs. This feature is the most important capability of a priority scheduler.

The medium-priority jobs show interesting behavior as well in Figure 6(b). Here, a sharp rise in job time occurs in the medium-priority chart just as the high-priority trends for the SJS and the PJS of Figure 6(a) are converging. The convergence happens after approximately 30 high-priority jobs have been scheduled. This rise can be explained by the fact that, at this point, the PJS has exhausted all of the clusters in its unfettered desire to optimize the high-priority jobs. The medium-priority jobs then are receiving the negative consequences resulting from that high-level decision and thus generally need to wait to run on the better machines. Shortly after this point, the high-priority jobs themselves can no longer benefit in the aggregate from parallel execution and thus more are scheduled as sequential jobs. This decision benefits the high-priority jobs slightly, but it benefits the medium-priority jobs greatly, as shown in the sharp fall in their total job times. The same behavior is evidenced in Figures 6(e) and 6(h), but it occurs earlier in the latter chart because high-priority jobs outnumber medium-priority jobs in the 50/25/25 distribution.

The performance of the low-priority jobs is shown in Figures 6(c), 6(f), and 6(i). In Figures 6(c) and 6(f), the PJS realizes overall performance marginally inferior to that of the SJS. The degradation of low-priority job performance relative to the SJS is not unexpected, because the PJS makes use of its ability to execute some of the higher priority jobs in parallel. This scheduling decision naturally leads to longer delays in obtaining the necessary resources for low-priority jobs relative to sequential job scheduling. Figure 6(i) shows that the performance of the PJS is comparable to the SJS under the 50/25/25 priority distribution. This behavior results from the fact that the high-priority jobs are numerous enough in this distribution to block their own parallel execution most of the time. Thus, the low-priority jobs are in roughly the same situation for this distribution in the PJS as they are in the SJS.

The data in Figure 7 was collected by computing the sum of the job times of all the jobs (i.e., at every priority level) for each number of total jobs considered. The mean of these summations was then displayed as a bar in the chart. Thus, the chart displays the mean over the entire job range of the summation of job times for each priority distribution and scheduler type (i.e., PJS and SJS).

Figure 7 shows that in terms of total performance (aggregating all priority levels), the priority scheduling systems are generally, but not always, slightly worse than the systems would have been without the priority schemes. It should be noted that both the PJS and the SJS algorithms are heuristic in nature and do not necessarily produce optimal schedules. There is a certain amount of noise in the scheduling performance metric based on how well the algorithm did in finding a minimum for each particular case. For small job sets, the algorithms generally come closer to optimality than for large job sets. Thus, because priority scheduling reduces the effective size of the scheduling problem (i.e., by dividing it into three problems), the efficiency of the algorithms themselves is slightly enhanced. This enhancement appears to partially offset the adverse effect imposed by the priority system due to the fact that the jobs at each priority level are scheduled without any knowledge of the jobs at lower priority levels. The tension between this intentional ignorance and the improved efficiency of the scheduling algorithm for smaller problem sizes explains why the priority scheduling systems generally, but not always, have an aggregate performance degradation that is somewhat less than 10%. As seen in Figure 7, there does not appear to be a significant difference between the PJS and the SJS in terms of aggregate overall performance in this priority experiment. This observation indicates that the PJS improves the performance of higher-priority jobs relative to the SJS by approximately the same amount as it worsens the performance of the lower-priority jobs relative to the SJS.

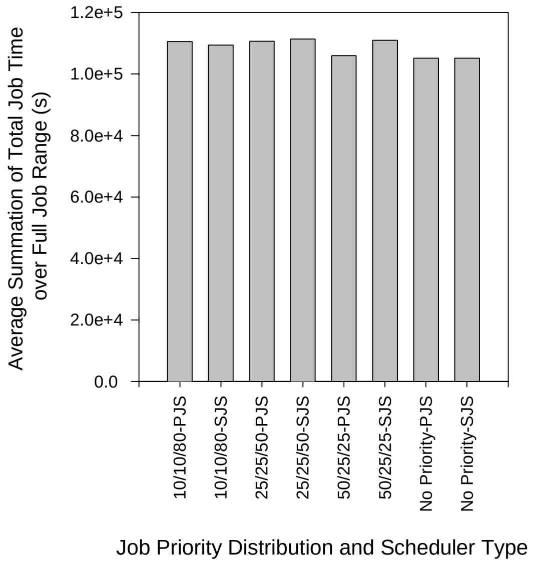

Figure 7. Comparison of average total performance of all jobs by scheduler type and priority distribution.

In summary, it can be concluded that the PJS is well suited for priority scheduling environments, as it effectively favors the high-priority jobs relative to its sequential ancestor. A determination of priority scheduling policy should also consider the performance penalties evidenced in aggregate performance over all priority levels. Finally, it should be noted that the SJS and the PJS have approximately the same aggregate performance penalty under priority scheduling, but that penalty is concentrated more on the low-priority jobs when the PJS is used.

5.4. Case Study 4: Preemptive Priority Scheduling Experiment with SWIL Simulation

The fourth and final experiment determines the performance impact of the use of a preemptive priority scheme on the PJS. Preemption is a moot issue when all the jobs in a job set are initially provided to the scheduler, so it was necessary to simulate the jobs as arriving at some average rate. In this experiment, high-priority jobs preempt low-priority jobs if the scheduler covets their assigned resources. The low-priority jobs are then rescheduled and no incremental progress is assumed. Medium-priority jobs are immune to preemption but cannot initiate preemption against low-priority jobs. Simulation makes this experiment possible and a Poisson arrival process for jobs was chosen. Several different mean job arrival rates are used (i.e., 1, 2, 4, 8, 16, and 32 jobs on the average per time quantum of 10 seconds). The job priority distributions used are the same as in the previous case study. The SWIL simulation environment for this case can be viewed as a superset of the other cases. For instance, if we set the mean arrival rate such that all of the jobs will arrive in the first time quantum, the problem reduces to that of the third case study on priority scheduling. Furthermore, if we also set the priority distribution such that all of the jobs have equal priority, the problem reduces to that of the second case study on job scheduling performance. Thus, the SWIL simulation environment prepared for this experiment is suitable for a broad range of possible experiments. The SWIL simulation environment in this case provides the jobs to the scheduler at an average rate, with priority levels randomly drawn from the underlying distribution, and with the job types drawn from the job type distribution in a random fashion as well. This task would be difficult to achieve faithfully without simulation, and the flexibility gained makes the environment highly reusable for future experiments and revised scheduling systems.

This final case study analyzes the effect of implementing a preemptive priority scheme on the performance of the PJS. In this experiment, 500 jobs were simulated using SWIL as arriving randomly according to each Poisson arrival rate used and with priority levels randomly determined using each priority level distribution. The experiment continued until all jobs had arrived and were completed. The mean job time for each job priority level for the 10/10/80 distribution is shown by arrival rate in Figure 8(a). Recall that job time is the difference between the time of final job completion and the time of first job submission. Figures 8(b) and 8(c) show the same information for the 25/25/50 and 50/25/25 priority level distributions, respectively.

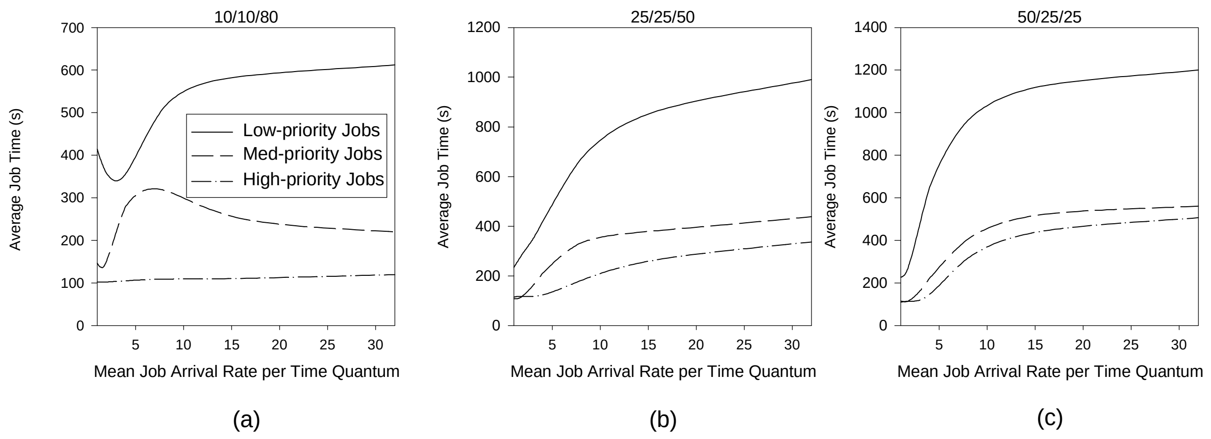

Figure 8. Comparison of average job times by priority level versus arrival rate for 10/10/80 priority distribution (a), 25/25/50 priority distribution (b), and 50/25/25 priority distribution (c).

High-priority status for jobs significantly reduces their mean total job time for all of the priority level distributions in Figure 8. As expected, the degree to which the average job time for high-priority jobs is reduced is directly related to the rarity of such jobs. For instance, the 10/10/80 distribution of Figure 8(a) is able to contain the average job time for its high-priority jobs to slightly over 100 seconds, whereas the high-priority jobs in the 50/25/25 priority distribution take on average over four times as long to complete under conditions of high arrival rate. The high rates of parallel execution (analyzed later in this section via Table 1) for high-priority jobs explain their high performance relative to jobs with lower priority. Furthermore, the medium-priority jobs also experience performance that is significantly superior to that of low-priority jobs. These observations show that the priority scheme well serves the expressed goals in implementing preemptive priority.

In all of the charts of Figure 8, it is observed that an increasing arrival rate generally increases the average job times. This statement is true in general because contention for resources increases with increasing arrival rates. However, the trends are not monotonic increasing for two reasons. One reason is that the more jobs available to the scheduler at one scheduling time, the more efficient the scheduling algorithms can be in resource allocation. This scheduling myopia has its greatest impact for slow arrival rates. Also, when the jobs come in slowly, there is a strong tendency for low-priority jobs to be preempted well into their execution. This effect is strongest for the 10/10/80 distribution for low arrival rates.

The average job time aggregated over all priority levels for each of the four priority-level distributions is depicted in Figure 9(a). Here, using the preemption scheme in conjunction with priority generally results in a slight degradation in overall performance relative to the no-priority case. For low arrival rates, the priority distributions that make more heavy use of the higher priorities do better in the aggregate whereas the trend reverses after passing a mean arrival rate of 8 jobs per time quantum.

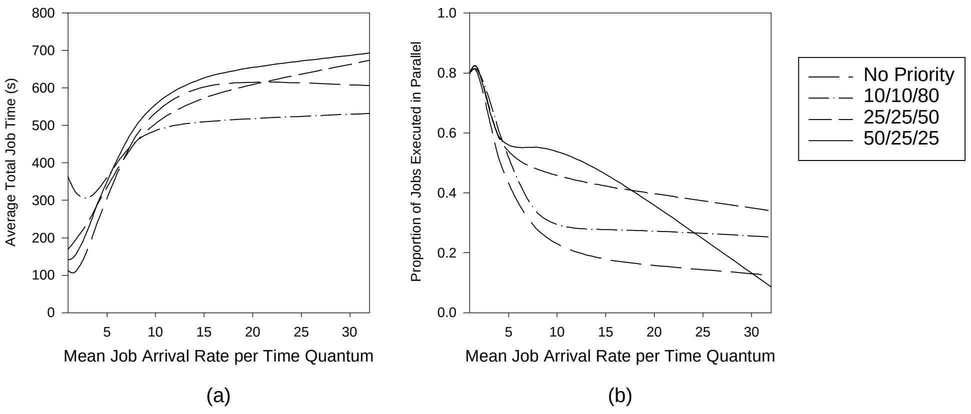

Figure 9. Comparison of average performance of priority distributions using preemptive priority (a), and proportion of jobs executed in parallel compared by priority distribution (b).

At low job arrival rates, the high-priority jobs in the 10/10/80 distribution are frequently preempting the low-priority jobs that are close to finishing. This frequent preemption occurs because the high-priority jobs in the 10/10/80 distribution and a mean job arrival rate of one job per time quantum appear every 100 seconds on average, whereas low-priority jobs occur eight times more often. Thus, the low-priority jobs are likely to be able to make a good amount of progress towards completion before a high-priority job appears. The arriving high-priority job is very likely to covet those resources being used by the low-priority jobs. This contention occurs because when the job arrival rate per time quantum is low, jobs tend to be concentrated on the most effective resources available. For the 25/25/50 and 50/25/25 distributions, this effect is less prominent because the effective arrival rate of high-priority jobs is greater and that of low-priority jobs is lesser, meaning that the low-priority jobs make less progress on average before being preempted. As the job arrival rate per time quantum increases, the impact of preempting low-priority jobs after substantial progress decreases, as the low-priority jobs are both coerced into sequential execution (analyzed later in this section via Table 1) and relegated to the least preferred machines as can be observed by their high average job times.

When the mean job arrival rate per time quantum is high, the smaller overall number of high-priority jobs in the 10/10/80 case means that the scheduler can be more efficient than in the 25/25/50 and 50/25/25 cases. This efficiency results from the fact that fewer constraints due to the scheduling of jobs with higher priority are being applied to the lower priority jobs than in the other cases. The improvement in scheduling efficiency begins to dominate the effect of frequent preemption after the mean arrival rate passes 8 jobs per time quantum, thus explaining why the 10/10/80 case experiences performance superior to the 25/25/50 and 50/25/25 cases when jobs arrive quickly.

The proportion of parallelism (i.e., where 0% means no jobs are executed in parallel, and 100% means all jobs are parallel) for each case is shown in Figure 9(b). In general, parallelism decreases as the arrival rate increases.

Table 1. Proportion of parallelism, by distribution, priority level, and rate of job arrival.

	Mean Job Arrival Rate per Time Quantum					
Priority Distribution, Priority Level	1	2	4	8	16	32
10/10/80, High-priority Jobs	82%	82%	82%	78%	78%	70%
10/10/80, Medium-priority Jobs	72%	72%	56%	46%	54%	64%
10/10/80, Low-priority Jobs	81%	78%	47%	20%	6%	1%
25/25/50, High-priority Jobs	78%	78%	76%	47%	46%	46%
25/25/50, Medium-priority Jobs	82%	82%	70%	46%	46%	46%
25/25/50, Low-priority Jobs	80%	79%	47%	18%	7%	2%
50/25/25, High-priority Jobs	80%	80%	65%	52%	48%	40%
50/25/25, Medium-priority Jobs	80%	80%	67%	65%	60%	57%
50/25/25, Low-priority Jobs	80%	80%	37%	23%	12%	8%
No Priority Levels, All Jobs	80%	80%	59%	55%	44%	9%

The proportion of parallelism is shown in Table 1 for each priority level, priority distribution, and job arrival rate. It is observed that the proportion, as it does with increased arrival rate, also decreases with a drop in the priority level of the job. Since the jobs with higher priority have the first call on the clusters, this behavior is an expected result. As the contention for resources rises, the PJS automatically reduces the proportion of jobs that it schedules to clusters rather than sequential machines. The proportion of parallelism falls faster with increasing job arrival rate for high-priority jobs in priority distributions with a higher percentage of such jobs. This effect is a clear consequence of the fact that high-priority jobs are less “special” in distributions that allocate such status more frequently. However, there are several confounding factors. If a job is subjected to frequent preemption, as with low-priority jobs with small arrival rates for the 10/10/80 case, when it finally does get to execute, it will usually get a cluster. Also, for low effective arrival rates, there is little difference between high- and medium-priority jobs in terms of proportion of parallelism, as there are enough cluster resources for both sets of jobs, although the best clusters are usually allocated to the high-priority jobs, as is evidenced by their lower average job times. Finally, the relative abundance of high-priority jobs in the 50/25/25 case at high arrival rates reduces the proportion of such jobs executed in parallel. This effect occurs because a large number of high-priority jobs often appear for scheduling during one time quantum under these conditions, causing the PJS to schedule fewer of them to clusters.

In summary, addition of preemption to the priority scheme does have a slight negative impact on overall job performance of the PJS. This negative impact is highly dependent on the arrival rate and the priority distribution. Overall performance degradation due to the use of preemptive priority is most pronounced for the combinations of low arrival rates and the 10/10/80 priority distribution, and for high arrival rates and the 25/25/50 and 50/25/25 priority distributions. Overall, preemption greatly benefits the high-priority jobs by giving them greater access to cluster resources for parallel execution and by reducing the amount of delay in obtaining the needed resources for execution. Since a preemptive priority job scheduler is principally judged by the performance of its high-priority jobs, it is clear that the PJS is effective in this role.

6. Conclusions

This paper provides an overview of the application of the SWIL simulation technique for the analysis and evaluation of a novel job scheduling algorithm intended to support parallel as well as sequential jobs in a heterogeneous, cluster computing environment. This approach uses the software implementation of a scheduling algorithm running in a partially or wholly simulated environment of computing resources and submitted jobs. Thereby, scheduling system performance is investigated under a range of conditions that were not feasible to duplicate with entirely real resources. Results are validated by the use of actual execution experiments on representative machines for each machine class in the resource set. The usefulness of this SWIL approach for the analysis of scheduling algorithms is then demonstrated with a series of four case studies that explore the performance and overhead of the PJS algorithm representing a significant evolution of those in the literature.

The first case study demonstrates that the scheduling algorithms analyzed have modest overheads, even when running on a fairly low-end machine. For instance, the SJS has a scheduling overhead of less than 0.2 seconds for a schedule involving 100 jobs and 100 machines. Further, the SWIL simulations show that the PJS imposes from two to three times the overhead of the SJS, and that this overhead is largely independent of the percentage of jobs that were candidates for parallel execution.

The second case study shows that the total execution time of a set of jobs can be reduced by up to 50% through the use of the PJS algorithm over that achieved by the SJS algorithm. This result is particularly notable in that the sequential job scheduling algorithm has excellent performance when compared to OLB and other RMS scheduling algorithms detailed in the literature. This performance improvement is always nonnegative as well, and is concentrated in regions of light job load, making the PJS algorithm a useful evolution of the SJS algorithm developed and employed in the *SmartNet* effort.

The effect of imposing a priority scheduling regime is examined in the third case study. It is demonstrated that the PJS algorithm has significantly more ability to favor high-priority jobs than the SJS algorithm. In addition, performance in the aggregate (i.e., considering all priority levels) is analyzed and the adverse effect of nonpreemptive priority scheduling is shown to be fairly minor under the conditions studied.

The fourth and final case study investigates the effect of imposing a preemptive priority scheme on the scheduling algorithms under varying conditions of job arrival rate and priority distribution. Under these conditions, SWIL testing proves that the PJS algorithm used in a preemptive priority mode of operation is very effective at favoring the high-priority jobs over the lower priority jobs as is the objective for a preemptive, priority-based scheduling system. There does appear to normally be a performance penalty in the aggregate sense associated with the use of preemption and priority under these conditions, but the penalty is not generally very large and in many cases the improved performance of higher priority jobs may be well worth the cost.

Several future directions to this research are possible using the SWIL simulation approach to the investigation of scheduling algorithms and RMS implementations. One possible direction is to investigate the effect of modifying the PJS algorithm as its heuristic nature indicates that there may well be room for improvement on both its performance and on its scheduling overhead. The development and analysis of analogous extensions for parallel job scheduling to some of the more exotic *SmartNet* scheduling algorithms, such as Genetic Simulated Annealing (GSA), might be one such possibility. Another possible direction of research is to further investigate the possibilities and trade-offs in a preemptive scheduling scheme. For instance, the effect of adding provisions to help prevent job starvation, such as increasing the priority level of a job after a certain number of preemptions, could be investigated. Another possible direction would be to examine the effect of modeling job execution time as a statistical distribution instead of a known quantity, as is the case in [2]. Yet another possibility is to investigate making the preemptive version of the PJS only preempt low-priority jobs if it sees a potential gain greater than some threshold. This change would increase scheduling overhead, but might reduce unnecessary contention over resources. A final possible area of future research would involve linking the SWIL approach to one of the many simulation tools available rather than the object-oriented C++ code that is currently used to simplify the development of experiments, and thereby produce a more integrated SWIL simulation environment.

Acknowledgments

This work was sponsored in part by the U.S. Department of Defense.

References

[1] J. Adams, D. Thomas, “Design Automation for Mixed Hardware-Software Systems,” *Electronic Design*, vol 45, n 5, Mar. 3, 1997, pp. 64-72.

[2] R. Armstrong, D. Hensgen, T. Kidd, “The Relative Performance of Various Mapping Algorithms is Independent of Sizable Variations in Run-time Predictions,” *Proceedings of the 7th Heterogeneous Computing Workshop*, 1998.

[3] D. Bailey, J. Barton, et al., “The NAS Parallel Benchmarks,” Moffet Field, California: NAS Systems Division, NASA AMES Research Center, 1991.

[4] G. Barlas, “Collection-Aware Optimum Sequencing of Operations and Closed-Form Solutions for the Distribution of a Divisible Load on Arbitrary Processor Trees,” *IEEE Transactions on Parallel and Distributed Systems*, vol 9 n 5, May 1998, pp. 429-441.

[5] P. Brucker, *Scheduling Algorithms*, second revised and enlarged edition, Springer, 1998.

[6] J. Buck, S. Ha, et al., “Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems,” *Int. Journal of Computer Simulation, special issue on "Simulation Software Development"* vol 4, April 1994, pp. 155-182.

[7] R. Buyya, “PARMON: A Portable and Scalable Monitoring System for Clusters,” *Software—Practice and Experience*, vol 30 n 7, Jun. 2000, pp. 723-739.

[8] D. Collins, “Integrating the Linda Distributed Computing Environment into the SmartNet Resource Management System,” M.S. Thesis in Electrical Engineering, Florida State University, Fall 1995.

[9] S. Darbha, D, Agrawal, “A Task Duplication Based Scalable Scheduling Algorithm for Distributed Memory Systems,” *Journal of Parallel and Distributed Computing*, vol 46 n 1, Oct. 10, 1997, pp. 15-27.

[10] X. Du, X., Zhang, “Coordinating Parallel Processes on Networks of Workstations,” *Journal of Parallel and Distributed Computing*, vol 46 n 2, Nov. 1, 1997, pp. 125-135.

[11] R. Freund, H. Siegel, “Heterogeneous Processing,” *IEEE Computer*, vol 26 n 6, June 1993, pp. 13-17.

[12] R. Freund, T. Kidd, et al., “The SmartNet Heterogeneous Scheduling Framework.” *Proceedings of the 2nd International Symposium on Parallel Architectures, Algorithms, and Networks*, 1996, pp. 514-521.

[13] A. George, R. Fogarty, et al., “An Integrated Development System for Parallel and Distributed System Prototyping,” *Simulation*, vol 72 n 5, May 1999, pp. 283-294.

[14] T. Hagerup, “Allocating Independent Tasks to Parallel Processors: An Experimental Study,” *Journal of Parallel and Distributed Computing*, vol 47 n 2, Dec. 15, 1997, pp. 185-197.

[15] B. Hamidzadeh, L. Kit, D. Lilja, “Dynamic Task Scheduling Using Online Optimization,” *IEEE Transactions on Parallel and Distributed Systems*, vol 11 n 11, Nov. 2000, pp. 1151-1163.

[16] C. Hui, S. Chanson, “Allocating Task Interaction Graphs to Processors in Heterogeneous Networks,” *IEEE Transactions on Parallel and Distributed Systems*, vol 8 n 9, Sep. 1997, pp. 908-925.

[17] K. Keeton, S. Rodrigues, D. Roselli, “Previous Work in Distributed Operating Systems,” Berkeley NOW White Paper, 1995.

[18] H. Koenignmann, G. Gurevich, “ATTSIM, Attitude Simulation with Control Software in the Loop,” *Proceedings of the 13th AIAA/USU Conference on Small Satellites*, 1999.

[19] Y. Kwow, I. Ahmad, “Efficient Scheduling of Arbitrary Task Graphs to Multiprocessors Using a Parallel Genetic Algorithm,” *Journal of Parallel and Distributed Computing*, vol 47 n 1, Nov. 25, 1997, pp. 58-77.

[20] J. Li, H. Kameda, “Load Balancing Problems for Multiclass Jobs in Distributed/Parallel Computer Systems,” *IEEE Transactions on Computers*, vol 47 n 1, Mar. 1998, pp. 322-332.

[21] Y. Li, J. Antonio, et al., “Estimating the Distribution of Execution Times for SIMD/SPMD Mixed-Mode Programs,” *Proceedings of the Heterogeneous Computing Workshop*, 1995, pp. 35-46.

[22] Message Passing Interface Forum, “Document for standard message-passing interface,” Technical Report CS-93-214, University of Tennessee, November 1993.

[23] M. Pallis, J. Liou, D. Wei, “Task Clustering and Scheduling for Distributed Memory Parallel Architectures,” *IEEE Transactions on Parallel and Distributed Systems*, vol 7 n 1, January 1996, pp. 46-55.

[24] M. Parashar, S. Hariri, “Interactive Performance Prediction for Parallel Application Development,” *Journal of Parallel and Distributed Computing*, vol 60 n 1, January 2000, pp. 17-47.

[25] Pfister, G., *In Search of Clusters*, Prentice Hall, December 1997.

[26] D. Ridge, D. Becker, et al., “Beowulf: harnessing the power of parallelism in a pile-of-PCs,” *Proceedings of the 1997 IEEE Aerospace Conference*, 1997, pp. 79-91.

[27] Scientific Computing Associates, *Linda Users Guide and Reference Manual, Release 3.1*, Scientific Computing Associates, Inc., Mar. 1995.

[28] V. Sunderam, “PVM: A Framework for Parallel Distributed Computing,” *Concurrency: Practice & Experience*, vol 2 n 4, December 1990, pp. 315-339.

[29] M. Tan, H. Siegel, et al., “Minimizing the Application Execution Time Through Scheduling of Subtasks and Communication Traffic in a Heterogeneous Computing System,” *IEEE Transactions on Parallel and Distributed Systems*, vol 8 n 8, August 1997, pp. 857-871.

[30] L. Wang, H., Siegel, et al., “Task Matching and Scheduling in Heterogeneous Computing Environments Using a Genetic-Algorithm-Based Approach,” *Journal of Parallel and Distributed Computing*, vol 47 n 1, Nov. 25, 1997, pp. 8-22.

[31] R. Wolski, N. Spring, “Application Level Scheduling of Gene Sequence Comparison on Metacomputers,” *Proceedings of the 12th ACM International Conference on Supercomputing*, 1998, pp. 141-148.

[32] R. Wolski, N. Spring, C. Peterson, “Implementing a Performance Forecasting System for Metacomputing: The Network Weather Service,” *Proceedings of the 11th ACM International Conference on Supercomputing*, 1997.

[33] A. Yan, J. Li, et al., “Determining the Execution Time Distribution for a Data Parallel Program in a Heterogeneous Computing Environment,” *Journal of Parallel and Distributed Computing*, vol 44 n 1, Jul. 10, 1997, pp. 35-52.

[34] A. Zomaya, M. Clements, S. Olariu, “A Framework for Reinforcement-Based Scheduling in Parallel Processor Systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol 9 n 3, Mar. 1998, pp. 249-260.