Stochastic Scheduling

Jennifer M. Schopf*
Computer Science Department
Northwestern University
jms@cs.nwu.edu

Francine Berman†
Dept. of Computer Science and Engineering
University of California, San Diego
berman@cs.ucsd.edu

August 9, 1999

Abstract

There is a current need for scheduling policies that can leverage the performance variability of resources on multi-user clusters. We develop one solution to this problem called **stochastic scheduling** *that utilizes a distribution of application execution performance on the target resources to determine a performance-efficient schedule.*

In this paper, we define a stochastic scheduling policy based on time-balancing for data parallel applications whose execution behavior can be represented as a normal distribution. Using three distributed applications on two contended platforms, we demonstrate that a stochastic scheduling policy can achieve good and predictable performance for the application as evaluated by several performance measures.

1 Introduction

Clusters of PCs or workstations have become a common platform for parallel computing. Applications on these platforms must coordinate the execution of concurrent tasks on nodes whose performance may vary dynamically due to the presence of other applications sharing the resources. To achieve good performance, application developers use performance models to predict the behavior of possible task and data allocations, and to assist in the selection of a performance-efficient application execution strategy. Such models need to accurately represent the dynamic performance variation of the application on the underlying resources in a manner which allows the scheduler to adapt application execution to the current system state.

In this paper, we discuss a strategy to use the variability of performance information in scheduling to improve application execution times. We focus in particular on data-parallel applications and represent the performance variation of the resources by normal distributions. Before describing the stochastic scheduling policy, we first motivate the use of stochastic (distributional) information for scheduling in the next subsection.

1.1 Stochastic Values

Performance models are often parameterized by values that represent system and/or application characteristics. In dedicated, or single-user, settings it is often sufficient to represent these characteristics by a single value, or **point value**. For example, we may represent bandwidth as 7 Mbits/second. However, point values are often inaccurate or insufficient representations for characteristics that change over time. For example, rather than a constant valuation of 7 Mbits/second, bandwidth may actually vary from 5 to 9 Mbits/second. One way to represent this variable behavior is to use a **stochastic value**, or distribution.

By parameterizing models with stochastic information, the resulting prediction is also a stochastic value. Stochastic-valued predictions provide valuable additional information that can be supplied to a scheduler and used to improve the overall performance of distributed parallel applications. Stochastic values can be represented in a variety of ways – as

* Partially supported by DARPA grant #N66001-97-C-8531.
† Supported in part by NSF grant #ASC-9701333, DARPA grant #N66001-97-C-8531.

distributions [SB97], as intervals [SB99], and as histograms [Sch99]. In this paper we assume that we can adequately represent stochastic values using normal distributions, as explained below.

This paper describes a methodology that allows schedulers to take advantage of stochastic information to improve application execution performance. We define a prototype scheduler that uses stochastic application execution time predictions to define a time-balancing scheduling strategy for data-parallel applications. This strategy adjusts the amount of data assigned to a processor in accordance with the variability of the system and the user's performance goals. We demonstrate that the stochastic scheduling approach promotes both good performance and predictable execution for three distributed applications on two multi-user clustered environments.

The paper is organized as follows: Section 2 describes a time balancing scheduling policy and shows how single values from stochastic parameters can be used to parameterize it. Experiments with this scheduling policy are described in Section 3. Related work is presented in 4, and we conclude in Section 5.

2 Time Balancing

For the purposes of this paper, we assume that the target set of resources is a fixed set of shared, heterogeneous single processor machines. We will consider a set of data parallel applications where the application is split into a number of identical "worker" tasks or a set of identical "worker" tasks and a "master" controller task, and where the amount of work each worker task performs is based on the amount of data assigned to it. Further, we assume that all communication and distribution of data is not overlapped with computation. Applications with this structure include sets of asynchronous iterations of iterated computations, and many master-worker calculations. Given these assumptions, scheduling simplifies to the data allocation problem: *How much data should be assigned to each processor, and how should this decision be made?*

Time balancing is a common scheduling policy for data parallel applications that attempts to minimize application execution time by assigning data to processors so that each processor finishes executing at roughly the same time. This may be accomplished by solving a set of equations, such as those given in Equation 1, to determine the set of data assignments $\{ D_i \}$

$$\begin{array}{l} D_i u_i + C_i = D_j u_j + C_j \ \ \forall\, i, j \\ \sum D_i = D_{Total} \end{array} \tag{1}$$

where

- u_i: Time to compute one unit of data on processor i.
- D_i: Amount of data assigned to processor i.
- D_{Total}: Total amount of data for the application.
- C_i: Time for processor i to distribute the data and communicate.

To solve these equations we assume that all variables are point-valued. We also assume that C_i is independent of the data assignment. (If the communication or data transfer time is a function of the amount of data assigned to a processor, this value can be added to the u_i factor.) An example of developing and solving time balancing equations to solve for a performance-efficient data allocation can be found in [BWF+96].

2.1 Stochastic Time Balancing

To allow for the use of stochastic information, we focus on the u_i parameter (time to compute one unit of data on processor i) in the time-balancing equations. In the non-stochastic setting, a single (point) value is provided for u_i and the equations are solved for the D_i. In the stochastic setting, we still use a single value for u_i (to allow for the solution of D_i) but choose that value from the range given by a stochastic parameter. This allows us the flexibility to choose larger or smaller point values of u_i in the range. The choice of a larger value of u_i results in an assignment of less data to processor i, possibly mitigating the effects of performance variability on the overall application behavior (by assigning less data to a high variability machine).

We make the assumption that the distribution of values for u_i can be adequately represented by a normal distribution. One reason we make this assumption is to be able to mathematically manipulate the stochastic values, since

normal distributions are closed under various arithmetic operations unlike most other distribution families. Also, when using normal distributions we can estimate that using a range around the mean of two standard deviations will capture approximately 95% of the values. Previous work has shown that using normal distributions to represent such behavior leads to good predictions of application behavior [SB97]. This can be the case even when the original data is not a normal distribution.

One approach to determine which value to choose for u_i is to consider the mean of the normal distribution plus or minus some number of standard deviations. Since the standard deviation is a value specific to the normal distribution representing the performance of a given machine, each processor could have its data allocation adjusted in a machine-dependent manner. We call the multiplier that determines the number of standard deviations to add to (or subtract from) the mean the **Tuning Factor**. The Tuning Factor can be defined in a variety of ways, and will determine the percentage of "conservatism" of the scheduling policy.

With this in mind, we can define u_i as

$$u_i = m_i + sd_i * TF \tag{2}$$

where

m_i**:** The *mean* of the normal distribution representing the predicted completion time for task/processor i.

sd_i**:** The *standard deviation* of the normal distribution representing the predicted completion time for task/processor i.

TF**:** The *Tuning Factor*, used to determine the number of standard deviations to add to (or subtract from) the mean value to determine how conservative the data allocation should be.

Given the definition of u_i in Equation 2, the set of equations over which we must find a solution is now:

$$\begin{array}{l} D_i(m_i + TF * sd_i) + C_i = D_j(m_j + TF * sd_j) + C_j \quad \forall\, i, j \\ \sum D_i = D_{Total} \end{array} \tag{3}$$

2.2 The Tuning Factor

In order to determine how to take into account the variability of the system into our scheduling decisions, we define a Tuning Factor. The Tuning Factor represents the variability of the system as a whole, and as such provides the "knob" to turn to make the scheduling policy more or less conservative. The Tuning Factor can be provided by the user or calculated by the scheduler. The challenge lies in choosing or calculating a Tuning Factor that promotes a performance-efficient schedule for a given application, environment, and user.

Our computational method for determining the Tuning Factor uses a system of benefits and penalties calculated using stochastic prediction information. This benefit and penalty approach to determining the value of the Tuning Factor draws from the scheduling approach of Sih and Lee [SL90b]. In their work, data assignments are increased for processors with desirable characteristics (such as a light load, fast processor speed, etc.), considered as **benefits**, and decreased for machines with undesirable properties (such as poor network connectivity, small memory capacity, etc.), considered as **penalties**. The basic idea behind our approach is to assign more optimistic scheduling policies to platforms with a smaller variability in performance.

We use three values to provide information about the variability of the system as a whole: the *number* of machines that exhibit fluctuating performance behavior in the platform; the variability of performance exhibited of each machine, represented as the standard deviation, sd of a stochastic prediction; and a measure of the available computing capacity of the machines, called *power* below. We want to benefit (give a more generous allocation to) machines and platforms that have a smaller performance variability associated with them. Specifically, we want to benefit:

Platforms with fewer varying machines. If only a few machines on a platform have widely varying behavior, the platform should have a more optimistic (less conservative) schedule than another platform that has the majority of its machines exhibiting widely varying behavior.

Low variability machines, especially those with lower powers. We want to benefit a platform more for having a slow machine with a high variability than having a fast machine with a high variability. The faster machine may have more data assigned to it, so the high variability is likely to have a greater impact on the overall application execution time.

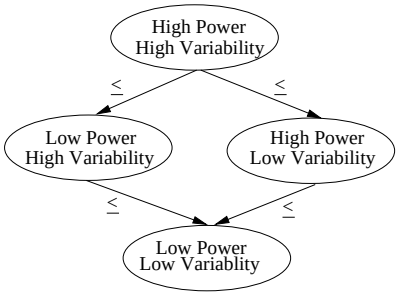

Figure 1. Diagram depicting the relative values of Tuning Factors for different configurations of power and variability.

Figure 1 provides an illustration of the relative Tuning Factor values for different configurations of machine power and variability. In Figure 2 we define an algorithm to calculate the Tuning Factor for stochastic values that obeys the partial ordering in Figure 1. Note that other Tuning Factor functions may be defined that satisfy the relative ranking in Figure 1.

```
For each task/processor pair, TP_i
   if Power (TP_i) > HighPower
      if Varibility(TP_i) > HighVariability
         Value_i = MaxTuningFactor
      else
         Value_i = MidTuningFactor
   else
      if Varibility(TP_i) > HighVariability
         Value_i = MidTuningFactor
      else
         Value_i = MinTuningFactor

TuningFactor =    Σ Value_i
                number of machines
```

Figure 2. Algorithm to Compute Tuning Factor.

The algorithm given in Figure 2 requires the definition of several parameters that are specific to the execution environment:

Power(TP$_i$): A measure of the communication and computational power of the machine with respect to the resource requirements of the application. This can be determined by an application-specific benchmark.

HighPower: A value that identifies machines with "high" power. For our initial experiments, we calculated an average power for the machines in the platform, and any machine with a greater value for power than the average was considered a High Power machine.

Variability(TP$_i$): A measure of the variability of performance delivered by a machine. For our experiments we set Variability(TP$_i$) equal to the standard deviation of the available CPU measurements for processor i.

HighVariability: A value that identifies when a machine has a high variability. There are many ways to chose this parameter. For our experiments, the CPU measurements generally fell into "modes" - approximately 0.5 when two processes were running, 0.33 when three were running, etc. We defined a HighVariability to be a standard deviation for the available CPU measurements greater than one-quarter of the width of the average mode from previous experimental data. This metric should be adjusted to the environment and can be defined by the user.

Tuning Factor Range: The Tuning Factor Range defines the range of values for the Tuning Factor. For the algorithm defined in Figure 2, three values within the Tuning Factor Range must be defined: *MinTuningFactor*, *MidTun-*

ingFactor, and *MaxTuningFactor*. These values can be used to determine the percentage of conservatism for the schedule that impacts the data allocation.

In this initial work, we focused on stochastic values represented using normal distributions and consequently focused on predictions at the mean, the mean plus one standard deviation, and the mean plus two standard deviations[1]. Therefore, we set *MinTuningFactor* = 0, *MidTuningFactor* = 1, and *MaxTuningFactor* = 2.

Note that using the algorithm given in Figure 2, schedules range only from a 50% to a 95% conservative schedule (i.e. a schedule which assigns less work processors whose performance exhibits high variance) as illustrated in Figure 3. This is because on a single-user (dedicated) platform, we expect mean values to achieve the best performance. Adding variability to the system (by adding other users) typically impacts performance in a negative way, leading to a more conservative schedule to mitigate the effect of the variability. The scheduling policy may also be adjusted to reflect other execution criteria such as socio-political factors that may affect application execution in a multi-user environment [LG97].

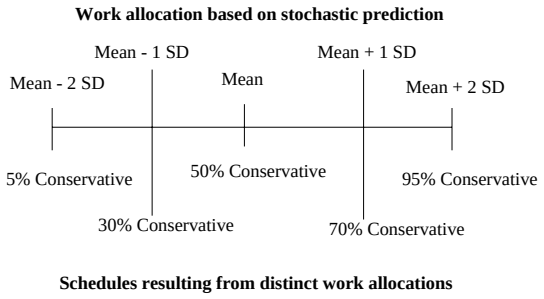

Figure 3. Diagram depicting the range of possible schedules given stochastic information.

3 Stochastic Scheduling Experiments

In this section we compare a time balancing scheduling policy parameterized by a mean-valued u_i (i.e. when the Tuning Factor is always 0) with two time balancing policies where u_i is determined by a non-zero Tuning Factor. We call the mean-valued scheduling policy (when TF=0) **Mean**. Mean represents a reasonable choice of a scheduling policy based on a fixed value to the time-balancing equations given in Equation 1 (we use the mean as it is the most likely point value representative of the normal distribution).

We compare Mean to two scheduling policies that use non-zero Tuning Factors. The first fixes the Tuning Factor to a 95% conservative schedule (where two standard deviations are added to the mean for the u_i value for every run), called **95TF**. The second determines a Tuning Factor at run-time based on environmental data calculated using the system of benefits and penalties described previously by the algorithm given in Figure 2, and is called the Variable Tuning Factor, or **VTF**. *Our goal is to experimentally determine in which situations it was advantageous to use non-zero Tuning Factors.*

All of the scheduling policies in our experiments use compositional structural performance prediction models [Sch97] parameterized by run-time information provided by the Network Weather Service [Wol96, Wol97, WSH98]. We ran the experiments on a set of applications using the data distribution determined by each policy at runtime and compared their execution times.

In order to compare the policies fairly we alternate scheduling policies for a single problem size of the application, so any two adjacent runs could experience similar workloads and variation in environment. (Each run was short, between 30 and 90 seconds.) For each set of experiments we show approximately 25 runs, but the experimental data is consistent with larger runs on similarly loaded platforms [Sch98].

[1] Given a normal distribution, a range around the mean plus or minus one standard deviation captures approximately 70% of the values and plus or minus two standard deviations captures approximately 95% of the values.

The experiments were run on UCSD's Parallel Computation Laboratory (PCL) and Linux clusters. The PCL cluster consists of 4 heterogeneous Sparc workstations connected by a mixture of slow and fast ethernet; the Linux Cluster consists of 4 PCs running Linux connected by fast ethernet. Both platforms and all resources were shared with other users during the time of the experiments. We define a machine with a **low** variation in CPU availability as having a variation of 0.25 or less, on a scale from 0 to 1, a **medium** variation when the CPU availability varied more than 0.25, but less than 0.5, and a **high** variation when the CPU values varied over half the range. A sample of the actual CPU values during the scheduling execution times are given for the runs shown in Figure 4 in graphs 5 through 8. For the rest of the experiments only summaries are given, although the full data sets may be found in [Sch98].

3.1 Performance Metrics

Each of the following graphs show the actual execution times using the three scheduling approaches, run one after the other so that each run will experience approximately equivalent system conditions. We consider a scheduling policy to be "better" than others if it exhibits the lowest execution time on a given run. In the experiments, the "best" scheduling policy varies over time and with different load conditions. To compare these policies, we define three performance metrics.

The first performance metric we define, called **Window**, is the proportion of runs by a scheduling policy that has the minimal execution time compared to the policy run before it and the policy run after it, or a window of three runs. For example, given the first few run times for the experiment in Figure 4, shown in Table 1, in the first set of three (Mean at 2623, VTF at 2689 and 95TF at 2758) the minimal time is achieved using the VTF policy. For the next set of three (VTF at 2689, 95TF at 2758 and Mean at 2821) VTF again achieved the minimal execution time, etc. Using this metric for the runs shown in Figure 4, the lowest execution time was achieved by Mean 9 times of 58 windows, for VTF 27 times of 58 windows and for 95TF 22 times of 58 windows, indicating that the VTF policy was almost three times as likely to achieve a minimal execution time than Mean.

Time stamp	Execution Time	Policy
2623	38.534076	Mean
2689	32.802351	VTF
2758	34.633066	95TF
2821	33.985560	Mean
2886	33.982391	VTF
2951	34.711265	95TF
3023	38.724127	Mean
3087	33.523844	VTF
3152	33.553239	95TF
3221	37.223270	Mean

Table 1. First 10 execution times for experiments pictured in Figure 4.

The second performance metric, which we call **Compare**, evaluates how often each run achieves a minimal execution time. There are three possibilities: it can have a *better* execution time than both the run before and the run after, it can have a *worse* execution time than both, or it can be *mixed* – better than one, and worse than the other. Referring to the execution times given in Table 1, for VTF at 2689, it achieves a better execution time than both the Mean run before it or the 95TF run after it; the 95TF run at 2758 does worse than either the VTF run before it or the Mean run after it, the Mean run at 2821 is mixed, it does better than the 95TF run before it, but worse than the VTF run after it. These results are given in Table 2 and indicate that VTF is three times as likely to have a faster execution time than the runs before or after it than the mean, and one-fourth as likely to be worse than both than the Mean for the experiments shown in Figure 4.

The third performance metric we use is a an **"average mean"** and an **"average standard deviation"** for the set of runtimes of each scheduling policy as a whole, as shown in Table 4. This metric gives a rough valuation on the performance of each scheduling approach over a given period of time. For example, for the data given in Figure 4, the Mean policy runs had a mean of 32.87 and a standard deviation of 3.780, the VTF policy runs had a mean 30.91 and a standard deviation of 3.34, and for the 95TF policy runs had a mean 30.66 and a standard deviation of 2.75. This

Experiment	Policy	Better	Mixed	Worse
Figure 4	Mean	3	4	12
	VTF	10	7	3
	95TF	6	9	4
Figure 9	Mean	8	6	5
	VTF	8	7	5
	95TF	3	6	10
Figure 10	Mean	3	7	9
	VTF	15	2	3
	95TF	3	8	8
Figure 11	Mean	8	4	7
	VTF	5	8	7
	95TF	6	8	5

Table 2. Summary statistics using Compare evaluation for SOR experiments.

indicates that over the entire run, the VTF policy exhibited a 5-8% improvement in overall execution time, and less variation in execution time than the Mean policy as well.

Experiment	Mean	VTF	95TF
Figure 4	9	27	22
Figure 9	24	24	10
Figure 10	8	39	11
Figure 11	25	14	19

Table 3. Window metric for each scheduling policy out of 58 possible windows for the SOR experiments.

Experiment	Mean		VTF		95TF	
	Mean	SD	Mean	SD	Mean	SD
Figure 4	32.87	3.78	30.91	3.34	30.66	2.75
Figure 9	34.52	3.63	34.52	1.22	35.23	1.58
Figure 10	24.74	2.72	22.15	2.98	24.48	2.57
Figure 11	23.31	4.35	22.78	2.74	22.31	2.75

Table 4. Average mean and average standard deviation for the set of runs for each scheduling policy for the SOR experiments.

3.2 Successive Over-Relaxation Code

The version of SOR (Successive Over-Relaxation) we use is a Regular SPMD code that solves Laplace's equation. Our implementation uses a red-black stencil approach where the calculation of each point in a grid at time t is dependent on the values in a stencil around it at time $t - 1$. The application is divided into "red" and "black" phases, with communication and computation alternating for each. In our implementation, these two phases repeat (and are time-balanced) for a set of 10 iterations.

Four experiments for the SOR code are shown in Figures 4 and 9 through 11. Figure 4 shows a comparison of the three scheduling policies on the Linux cluster when the available CPU on two machines had a low variation, on one had a medium variation and on the fourth had a high variation. Figure 9 shows a comparison of the three scheduling policies when there was a fairly constant low variance load on the Linux cluster as shown in Figures 5 through 8.

Figure 10 shows a comparison of the three scheduling policies when the PCL cluster had two machines with a very low variability, and two machines with high variability in available CPU. Figure 11 shows a comparison of the three scheduling policies when the PCL cluster had three machines with a high variability in available CPU. The **Compare**, **Window**, and **Average Mean** metrics are given for all environments in Tables 2, 3, and 4.

From Table 2, it is clear that different policies excel for different load conditions and that no one scheduling policy is the clear choice for all runs. However, the experiments do show that the scheduling policy in which the value of u_i is not fixed before execution (VTF) appears to perform better overall (achieve the lowest execution times) under a spectrum of load conditions. Moreover the conservative policy (95TF) appears to perform slightly better than Mean under these same conditions.

Values for the Window metric are given in Table 3. Using this metric, VTF also achieves the best performance on the whole although the ranking of the conservative policy 95TF and the Mean with respect to one another is inconclusive. Based on our experience with building AppLeS application schedulers [BWF+96, BW97], it is not surprising that adaptation to run-time load information promotes application performance. What is significant is that a straightforward calculation of a Tuning Factor using run-time information can be useful to the scheduler and promote performance-efficient application execution.

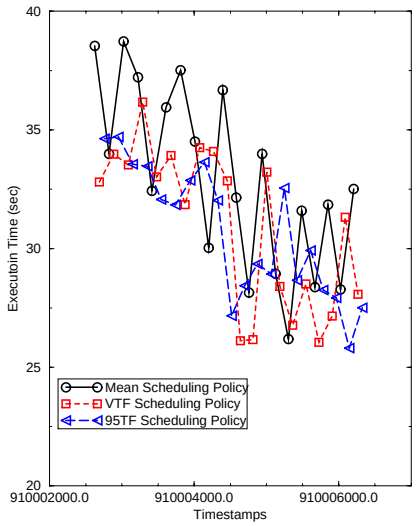

Figure 4. Comparison of Mean, VTF, and20 95TF policies, for the20 SOR benchmark on the Linux cluster with CPU loads shown in Figures 5 through 8.

3.3 Genetic Algorithm

We implemented a genetic algorithm (GA) heuristic for the Traveling Salesman Problem (TSP) [LLKS85, WSF89]. Our distributed implementation of this genetic algorithm uses a global population and synchronizes between generations [Bha96]. It was written in C using PVM. This application has a typical Assign Master-Worker structure. All of the workers operate on a global population (each member of the population is a solution to the TSP for a given set of cities) that is broadcast to them by the master using a PVM multicast routine. Each worker computes in isolation to create a specified number of children (representing new tours), and to evaluate them. All the sets of children are received by the master. They are sorted (by efficiency of the tour), some percentage are chosen to be the next generation, and the cycle begins again. In our implementation, a worker process runs on the master processor. Data assignment is done statically at runtime.

The GA has the added complication of nondeterministic behavior, making experimental comparisons difficult since not only the loads are changing but the actual work being done by the application changes as well. This is shown by Figure 12 which shows the run times for the GA on the dedicated Linux cluster when the data was distributed evenly

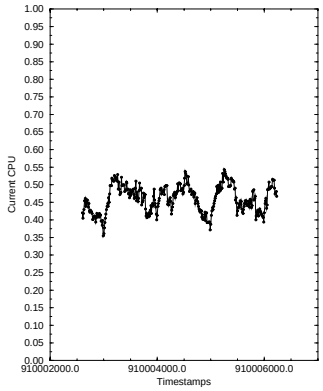

Figure 5. CPU values during runtime for scheduling experiments depicted in Figure 4 on Mystere, in the Linux cluster.

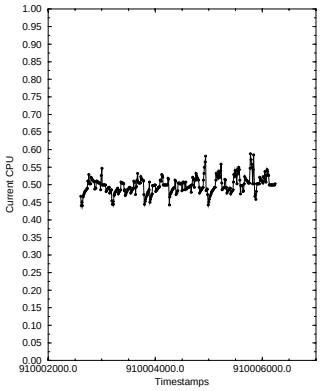

Figure 6. CPU values during runtime for scheduling experiments depicted in Figure 4 on Quidam, in the Linux cluster.

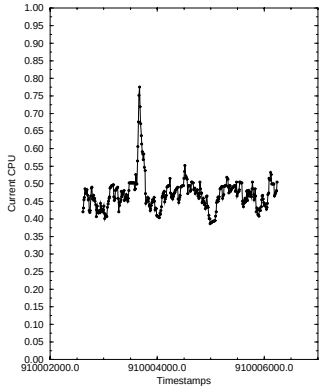

Figure 7. CPU values during runtime for scheduling experiments depicted in Figure 4 on Saltimbanco, in the Linux cluster.

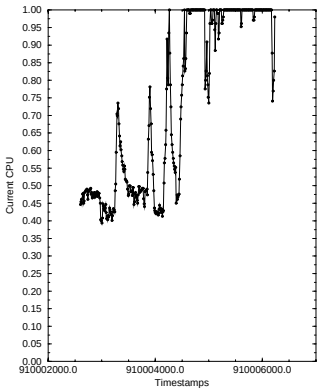

Figure 8. CPU values during runtime for scheduling experiments depicted in Figure 4 on Soleil, in the Linux cluster.

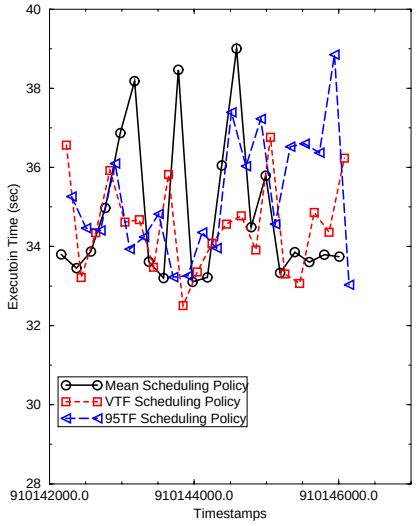

Figure 9. Comparison of Mean, VTF, and 95TF policies, for the SOR benchmark on the Linux cluster for 4 low variability machines.

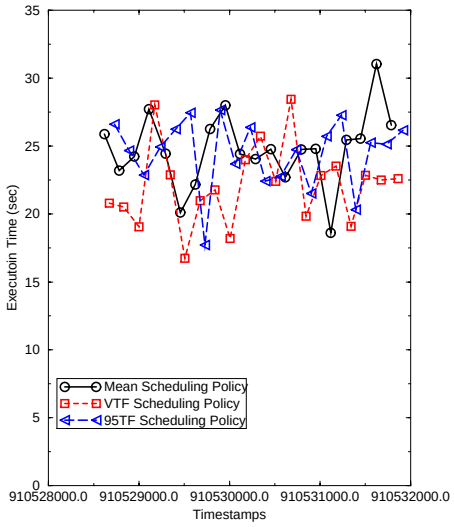

Figure 10. Comparison of Mean, VTF, and 95TF policies, for the SOR benchmark on the PCL cluster when 2 of the machines had a low variability in available CPU and two had high variability.

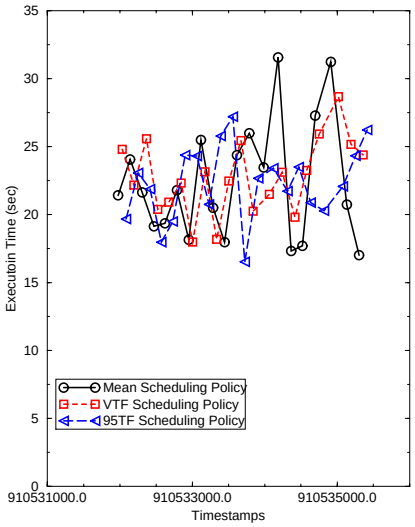

Figure 11. Comparison of Mean, VTF, and 95TF policies, for the SOR benchmark on the PCL cluster when 1 of the machines had a low variability in available CPU and three had high variability.

among the four processors, consistently from run to run. For these runs, the mean was 135.48 with a standard deviation of 5.04.

Experiment	Policy	Better	Mixed	Worse
Figure 13	Mean	9	3	7
	VTF	9	5	6
	95TF	6	3	10
Figure 14	Mean	5	5	9
	VTF	8	4	8
	95TF	9	5	5
Figure 15	Mean	4	8	8
	VTF	8	7	6
	95TF	7	7	6
Figure 16	Mean	2	10	8
	VTF	10	9	2
	95TF	4	10	6
Figure 17	Mean	10	6	4
	VTF	5	7	9
	95TF	5	7	8

Table 5. Summary statistics using Compare evaluation for GA experiments.

In Tables 5, 6, and 7 we give the Compare, Window and Average statistics, respectively, for the entire set of experiments for the Genetic Algorithm.

Figure 13 shows a comparison of the three scheduling policies when the Linux cluster had CPU loads that were high, but consistent, variability for two of the machines and low for the other two. The statistics indicate only a slight reduction in execution time, and less variance in execution time as well. Note that VTF performed better than the Mean scheduling policy on a run-by-run basis.

Figure 14 shows a comparison of the three scheduling policies when the Linux cluster had CPU loads in which two machines showed a very high variability in available CPU and the other two were low variability. Again, there

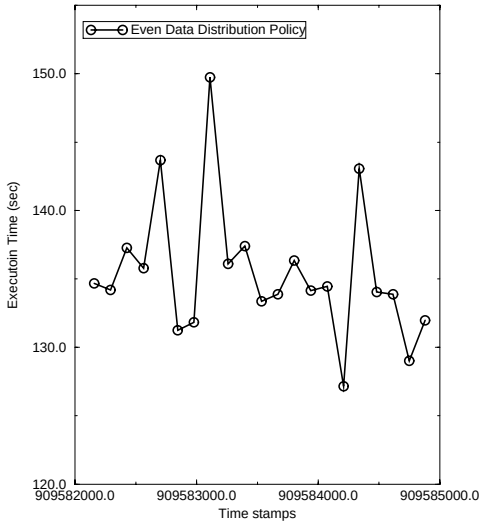

Figure 12. Run times for GA application with even data decomposition on Linux cluster.

Experiment	Mean	VTF	95TF
Figure 13	19	24	15
Figure 14	16	21	21
Figure 15	17	23	21
Figure 16	11	33	17
Figure 17	27	17	17

Table 6. Window metric for each scheduling policy out of 58 possible windows for GA experiments.

was a slight improvement in overall runtimes (2%), but the standard deviation was reduced by almost 40% when comparing Mean and 95TF. On a run-by-run basis, the stochastic policies were almost twice as likely to result in a faster application execution time.

Figure 15 shows a comparison of the three scheduling policies when the PCL cluster had CPU loads with two high variability and two moderate variability machines. Of note with this set of experiments is the spike seen at time 909960000 where both stochastic scheduling policies compensate for the variation significantly better than the Mean scheduling policy. Over the entire set of runs, the VTF policy showed about a 5% improvement over the Mean policy, and had a much smaller variation as well. The Compare metric also indicates that VTF was almost twice as likely to have the best execution time than Mean.

Figure 16 shows a comparison of the three scheduling policies when the PCL cluster had highly variable CPU loads for all four machines. It is unclear why 95TF performed so poorly, however VTF showed a 5% improvement over the mean execution times of the Mean policy. Likewise, VTF was almost 5 times more likely to have a faster execution time than Mean in a run-by-run comparison.

Figure 17 shows a comparison of the three scheduling policies when the PCL cluster had one highly variable, two moderately variable and one low variability machine. Of note with this set of experiments is that 95TF appears to be overly conservative when one machine had idle cycles, although VTF appears to adjust more efficiently as one would expect.

Experiment	Mean		VTF		95TF	
	Mean	SD	Mean	SD	Mean	SD
Figure 13	66.37	2.20	65.05	2.01	66.62	2.01
Figure 14	68.18	2.67	66.93	2.52	66.93	1.67
Figure 15	78.93	13.44	75.60	9.61	77.31	9.76
Figure 16	76.48	24.89	72.55	26.9	75.48	23.3
Figure 17	41.31	5.28	42.18	4.79	43.62	6.82

Table 7. Average mean and average standard deviation for entire set of runs for each scheduling policy for the GA experiments.

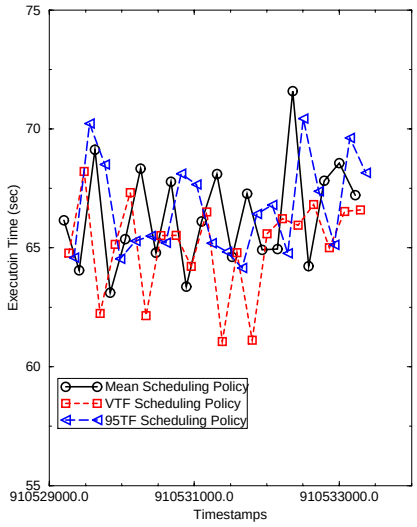

Figure 13. Comparison of Mean, VTF, and 95TF policies, for the GA code on the Linux cluster with two high and two moderately variable machines on the Linux cluster.

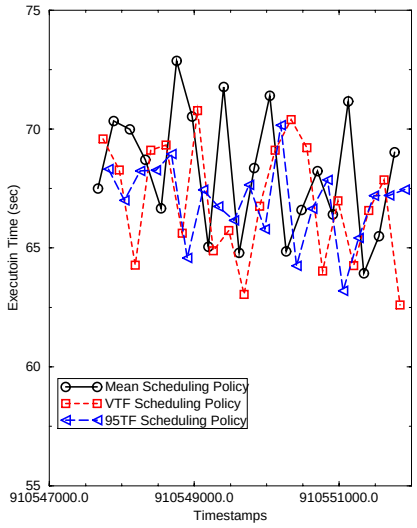

Figure 14. Comparison of Mean, VTF, and 95TF policies, for the GA code on the Linux cluster with two high and two low variable machines.

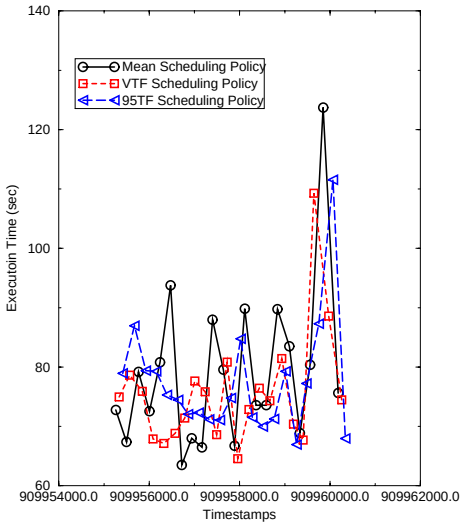

Figure 15. Comparison of Mean, VTF, and 95TF policies, for the GA code on the PCL cluster with two high and two moderately variable machines.

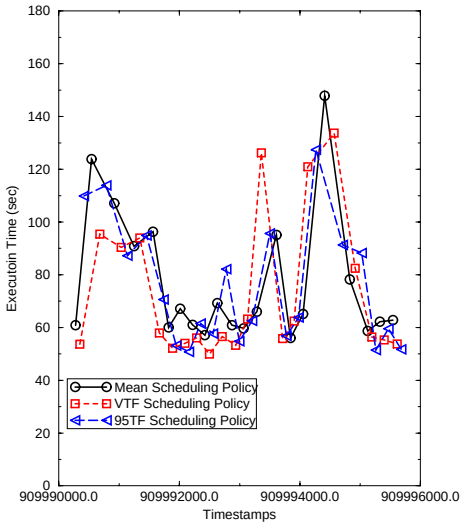

Figure 16. Comparison of Mean, VTF, and 95TF policies, for the GA code on the PCL cluster with four highly variable machines.

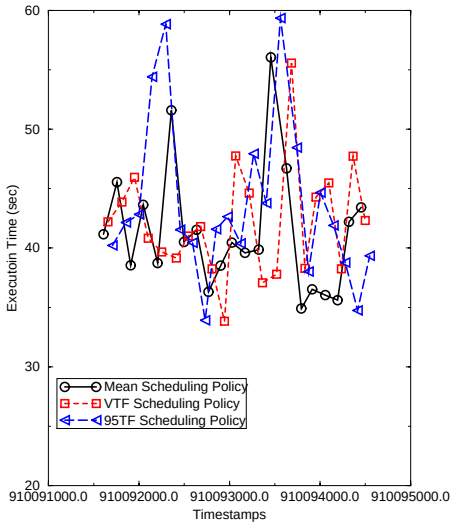

Figure 17. Comparison of Mean, VTF, and 95TF policies, for the GA code on the PCL cluster with one high, two moderate and one low variability machines.

3.4 N-body Code

We also ran experiments using an N-Body solver. N-body simulations, also known as particle simulations, arise in astrophysics and celestial mechanics (gravity is the force), plasma simulation and molecular dynamics (electrostatic attraction or repulsion is the force), and computational fluid dynamics (using the vortex method). These problems were some of the first problems addressed in parallel computation and continue to be studied in distributed parallel environments. The papers collected in the SIAM mini-symposium [HT97] and its predecessor [Bai95] offer ample evidence of the breadth and importance of N-body methods.

The problem our N-body implementation addresses is the tracking of the motion of planetary bodies. We scale the number of bodies involved, using a two-dimensional space with boundaries. Our implementation is an exact N^2 version that calculates the force of each planet on every other, and then updates their velocities and locations accordingly for each time step. It is based on an Assign-style Master-Worker approach [CI98] and written in C using PVM. For each iteration, the master sends the entire set of bodies to each worker, and also assigns a portion of the bodies to each processor to calculate the next iteration of values. The workers calculate the new positions and velocities for their assigned bodies, and then send this data back to the master process. Data decomposition is done statically at runtime by the master, which also reads in an initial data file. Unlike the GA, this code was deterministic, but communication played a larger role in predicting the execution time.

Experiment	Policy	Better	Mixed	Worse
Figure 18	Mean	5	7	7
	VTF	8	5	7
	95TF	7	5	7
Figure 19	Mean	4	6	9
	VTF	13	1	6
	95TF	6	5	8
Figure 20	Mean	3	8	8
	VTF	8	8	4
	95TF	7	6	6

Table 8. Summary statistics using Compare evaluation for N-body experiments.

Experiment	Mean	VTF	95TF
Figure 18	15	23	20
Figure 19	8	34	16
Figure 20	9	26	23

Table 9. Window metric for each scheduling policy out of 58 possible windows for N-body experiments.

Experiment	Mean		VTF		95TF	
	Mean	SD	Mean	SD	Mean	SD
Figure 18	61.93	7.48	59.84	5.22	59.73	4.81
Figure 19	53.96	4.46	52.48	5.42	54.49	5.75
Figure 20	62.64	10.03	59.93	11.72	60.49	10.05

Table 10. Average mean and average standard deviation for entire set of runs for each scheduling policy for N-body experiments.

Figure 18 shows a comparison of the three scheduling policies when the Linux cluster had one high, two moderate and one low variability CPUs. These statistics show a slight improvement in overall execution time (4-5%) when

comparing the stochastic policies to the Mean policy, and a reduced variation as well. Moreover, on a run-by-run basis, as shown by the Window and Compare metrics, the stochastic scheduling policies were more likely to achieve a better execution time.

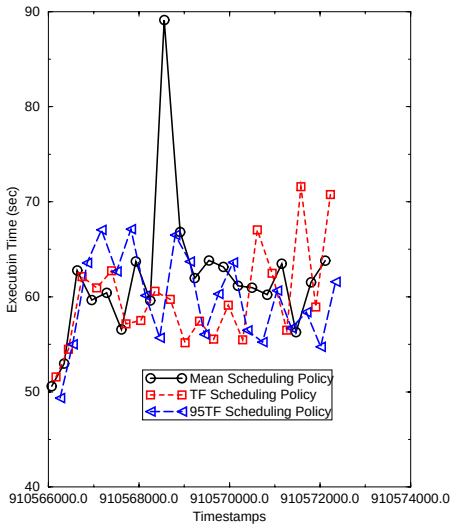

Figure 18. Comparison of Mean, VTF, and 95TF policies, for the NBody benchmark on the Linux cluster with one low, two moderate, and one high variability machines.

Figure 19 shows a comparison of the three scheduling policies when the Linux cluster had two low and two moderate variability CPUs. Using these statistics, we see only a slight improvement in overall execution times when comparing the Mean policy to the two stochastic policies, and not even that when comparing the Mean and 95TF. However, the Compare and Window metrics indicate that on a run-by-run basis, there is a significant improvement, with VTF having a minimal execution time nearly three times as often as Mean.

Figure 20 shows a comparison of the three scheduling policies when the Linux cluster had three highly variably and one low variability CPU. Using these statistics, we see that there is a 5% improvement when using VTF over the Mean policy, although for these runs there is not an improvement in the standard deviation of the execution times. Of note is the large change in performance seen at time stamp 910596000 when the load on three of the machines increased significantly.

3.5 Summary

The previous subsections presented graphs comparing three scheduling policies for several applications on two shared clusters of workstations. In general, we found that the policies with non-zero Tuning Factors (VTF and 95TF) led to reduced execution times when there was a high variation in available CPU. In addition, in almost all cases the standard deviation (or variation associated with the actual execution times) was greatly reduced when using non-zero Tuning factors, leading to more predictable execution time behavior.

4 Related Work

The most closely related work, and the primary work this project grew out of, is the AppLeS (Application Level Scheduling) Project [BW96, BW97, BWF+96]. The AppLes approach recognizes that applications can profitably use adaptive schedules to achieve the best possible performance. A custom AppLeS scheduling agent, integrated with the application, is developed to maximize the performance of the application using dynamic information and adaptive scheduling techniques.

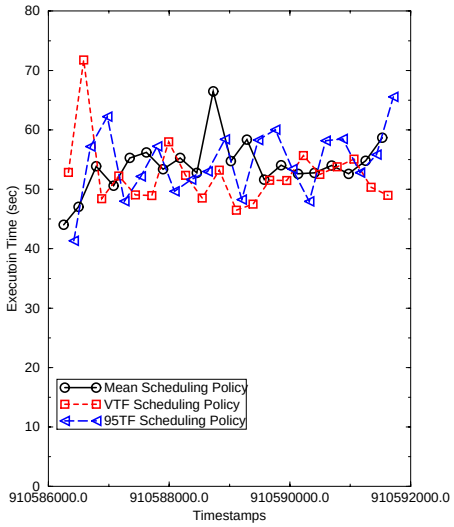

Figure 19. Comparison of Mean, VTF, and 95TF policies, for the NBody benchmark on the Linux cluster two low and two moderate variability machines.

Prophet [WZ98] uses run-time granularity information to select the best number of processors to apply to the application, and is also aware of the overhead possibly added by a scheduler. Prophet supports the use of highly-shared networks by utilizing dynamic system information about the level of CPU and network usage at runtime supplied by the Network resource Monitoring System (NRMS), much as our scheduling and prediction approach made use of the Network Weather Service.

Our approach to data allocation was based on a scheduling approach originally presented by Sih and Lee [SL90b, SL90a, SL93]. This work concentrated on a compile-time scheduling approach for interconnection-constrained architectures, and used a system of benefits and penalties (although not identified as such) for task mapping and data allocation. As a compile-time approach, it did not use current system information, and targeted a different set of architectures than our work as well.

Zaki et al. [ZLP96] also focused on the data allocation aspect of scheduling. This work compared load balancing techniques showing that different schemes were best for different applications under varying program and system parameters. However, the load model used in this work did not accurately portray the available CPU seen on our systems.

Waldspurger and Weihl have examined the use of a randomized resource allocation mechanism called *lottery scheduling* [WW94]. Lottery Scheduling is an operating system level scheduler that provides flexible and responsive control over the relative execution rates of a wide range of applications. It is related to our approach in that they manage highly shared resources using time series information, however this approach is not directly applicable to a higher level scheduler such as ours. In addition, this work allows modular resource management, an approach that may extend into our scheduling in the future.

Remulac [BG98] investigates the design, specification, implementation and evaluation of a software platform for network-aware applications. Such applications must be able to obtain information about the current status of the network resources, provided by Remos (REsource MOnitoring System) [LMS+98]. Remos and Remulac address issues in finding a compromise between accuracy (the information provided is best-effort, but includes statistical information if available) and efficiency (providing a query-based interface, so applications incur overhead only when they acquire information) much as we were concerned with the tradeoffs between accuracy and efficiency in stochastic information and scheduling choices.

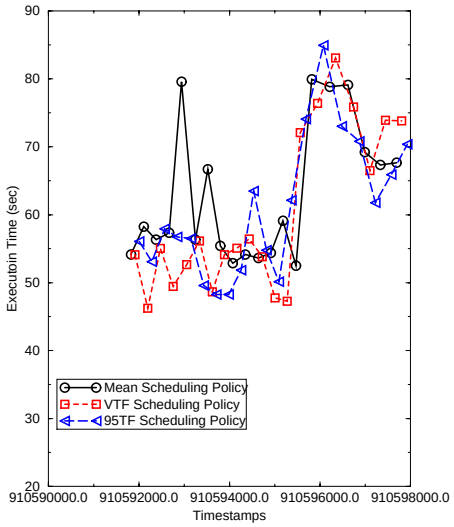

Figure 20. Comparison of Mean, VTF, and 95TF policies, for the NBody benchmark on the Linux cluster with three highly variable machines.

5 Summary and Future Work

In this paper we present an approach to stochastic scheduling for data parallel applications. We modify a time-balancing data allocation policy to use stochastic information for determining the data allocation when we have stochastic information represented as normal distributions. Our stochastic scheduling policy uses a Tuning Factor that determines how many standard deviations should be added based on run-time conditions. The Tuning Factor is used as the "knob" that determines the percentage of conservatism needed for the scheduling policy.

We illustrate this approach with the scheduling of a distributed SOR application, a Genetic Algorithm Application and an N-body calculation. Our experimental results demonstrate that it is possible to obtain faster execution times and more predictable application behavior (an important property for application developers) using stochastic scheduling.

There are many possible areas for future work. The approach discussed here is one of several ways to define the Tuning Factor, the heart of our stochastic scheduling policy. Other choices are possible (e.g. the impact of network performance could be included in the calculation of the Tuning Factor), and we plan to explore additional approaches that may better suit other particular environments and applications. Moreover, additional work must be done for environments in which performance cannot be adequately represented by normal distributions. In addition, stochastic scheduling policies must be defined for scheduling policies other than time-balancing and for additional classes of applications.

We believe that adaptive techniques that can exploit variance and other attributes of dynamic information will be critical to the development of successful applications in distributed multi-user environments. The stochastic scheduling results reported here are promising in that they demonstrate that dynamic and stochastic information can be used successfully to promote adaptation and improve application performance. We believe that they are a valuable first step in the process of developing adaptive scheduling strategies which promote the performance of distributed applications in these complex and challenging environments.

Acknowledgements

The Network Weather Service was developed by Rich Wolski, and supported by Neil Spring, Graziano Obertelli and Jim Hayes in its use. Many thanks to Rich and the rest of the AppLeS group as well for substantiative comments over the development of this work.

References

[Bai95] D. H. Bailey, editor. *Seventh SIAM conference on parallel processing for scientific computing*, 1995.

[BG98] J. Bolliger and T. Gross. A framework-based approach to the development of network-aware applications. *IEEE Transactions on Software Engineering (Special Issue on Mobility and Network-Aware Computing)*, 24(5), May 1998.

[Bha96] Karan Bhatia. Personal communication, 1996.

[BW96] Francine Berman and Richard Wolski. Scheduling from the perspective of the application. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, 1996.

[BW97] Francine Berman and Richard Wolski. The apples project: A status report. In *Proceedings of the 8th NEC Research Symposium*, 1997.

[BWF+96] Francine Berman, Richard Wolski, Silvia Figueira, Jennifer Schopf, and Gary Shao. Application-level scheduling on distributed heterogeneous networks. In *Proceedings of SuperComputing '96*, 1996.

[CI98] Alex Cereghini and Wes Ingalls. Personal communication, 1998.

[HT97] M. Heath and V. Torczon, editors. *Eighth SIAM conference on parallel processing for scientific computing*, 1997.

[LG97] Richard N. Lagerstrom and Stephan K. Gipp. Psched: Political scheduling on the cray t3e. In *Proceedings of the Job Scheduling Strategies for Parallel Processing Workshop, IPPS '97*, 1997.

[LLKS85] Lawler, Lenstra, Kan, and Shmoys. *The Traveling Salesman Problem*. John Wiley & Sons, 1985.

[LMS+98] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok. A resource query interface for network-aware applications". In *Proceedings of the 7th IEEE Smposium on High-Performance Distributed Computing*, 1998.

[SB97] Jennifer M. Schopf and Francine Berman. Performance prediction in production environments. In *Proceedings of IPPS/SPDP '98*, 1997.

[SB99] Jennifer M. Schopf and Francine Berman. Using stochastic intervals to predict application behavior on contended resources. In *Proceedings of the Workshop on Advances in Parallel Computing Models, ISPAN 99*, 1999. Also available as Northwestern University, Computer Science Department Technical Report CS-99-1, or `http://www.cs.nwu.edu/~jms/Pubs/TechReports/hilo.ps`.

[Sch97] Jennifer M. Schopf. Structural prediction models for high-performance distributed applications. In *CCC '97*, 1997. Also available as `www.cs.ucsd.edu/ users/jenny/ CCC97/index.html`.

[Sch98] Jennifer M. Schopf. *Performance Prediction and Scheduling for Parallel Applications on Multi-User Clusters*. PhD thesis, University of California, San Diego, 1998. Also available as UCSD CS Dept. Technical Report, Number CS98-607, `http://www.cs.nwu.edu/~jms/Thesis/thesis.html`.

[Sch99] Jennifer M. Schopf. A practical methodology for defining histograms in predictions. In *To Appear in ParCo '99*, 1999. Abstract available as Northwestern University, Computer Science Department Technical Report CS-99-2, or `http://www.cs.nwu.edu/~jms/Pubs/TechReports/hist.ps`.

[SL90a] Gilbert C. Sih and Edward A. Lee. Dynamic-level scheduling for heterogeneous processor networks. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Systems*, 1990.

[SL90b] Gilbert C. Sih and Edward A. Lee. Scheduling to account for interprocessor communication within interconnection-constrained processor networks. In *Proceedings of the 1990 International Conference on Parallel Processing*, 1990.

[SL93] Gilbert C. Sih and Edward A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2), 1993.

[Wol96] Rich Wolski. Dynamically forecasting network performance using the network weather service(to appear in the journal of cluster computing). Technical Report TR-CS96-494, UCSD, CSE Dept., 1996.

[Wol97] R. Wolski. Dynamically forecasting network performance to support dynamic scheduling using the network weather service. In *Proc. 6th IEEE Symp. on High Performance Distributed Computing*, August 1997.

[WSF89] D. Whitley, T. Starkweather, and D'Ann Fuquay. Scheduling problems and traveling salesman: The genetic edge recombination operator. In *Proceedings of International Conference on Genetic Algorithms*, 1989.

[WSH98] Rich Wolski, Neil Spring, and Jim Hayes. Predicting the cpu availability of time-shared unix systems. In *submitted to SIGMETRICS '99 (also available as UCSD Technical Report Number CS98-602)*, 1998.

[WW94] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First Symposium on Operating System Design and Implementation*, November 1994.

[WZ98] Jon B. Weissman and Xin Zhao. Scheduling parallel applications in distributed networks. *Journal of Cluster Computing*, 1998.

[ZLP96] Mohammed J. Zaki, Wei Li, and Srinivasan Parthasarathy. Customized dynamic load balancing for network of workstations. In *Proceedings of HPDC '96*, 1996.