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

May 27, 1999

Abstract

There is a current need for scheduling policies that can leverage the performance variability observed when scheduling parallel computations 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 adequately represented as a normal distribution. We demonstrate using a distributed SOR application 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 in which the performance variation can be adequately represented by a normal distribution. 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

*Partially supported by NASA GSRP Grant #NGT-1-52133, Darpa grant #N66001-97-C-8531.
†Supported in part by NSF grant #ASC-9701333, Darpa grant #N66001-97-C-8531.

and improve the overall performance of distributed parallel applications. In this paper we assume that we can adequately represent stochastic values using normal distributions, and that looking at a range around the mean of two standard deviations will capture approximately 95% of the values.

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 a distributed data-parallel SOR application[1] in 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 workstations. We will consider the set of data parallel applications where the application is split into a number of tasks, and the amount of work each task performs is based on the amount of data assigned to it. Given these assumptions, scheduling simplifies to the data allocation problem: *How much data should be assigned to each processor, and how should that decision be made?*

Time balancing is a common scheduling policy for data parallel applications that attempts to minimize the execution time of a data parallel application by assigning data 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 data assignments ($\{ D_i \}$ in the following equations).

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

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 to distribute the data.

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 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 value (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 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 the stochastic prediction of performance. This allows us the flexibility to choose larger or smaller 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

[1]In this extended abstract we focus only on SOR. Results for a distributed Genetic Algorithm and a distributed N-body code will be discussed in the final paper.

of performance variability on the overall application behavior (by assigning less data to a high variability machine).

Recall that we are assuming that the distribution of values for u_i can be adequately represented by a normal distribution.[2] 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 performance of a given machine, each processor would 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 predicted completion time for task/processor pair i.

sd_i: The *standard deviation* of the predicted completion time for task/processor pair 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

The Tuning Factor represents the variability of the system as a whole, defined by the amount of variation in the available CPU, the variation in the available bandwidth between each pair of processors, a measure of nondeterminism of the application, etc. The Tuning Factor is a key element in defining a stochastic time balancing policy. It 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 optimistic schedule to) the platforms that have a smaller variability associated with them. Specifically, we want to benefit:

Platforms with fewer varying machines. If only one machine on a platform has varying behavior, the platform should have a more optimistic schedule than another platform that has the majority of the machines exhibiting variable behavior.

[2] Previous work has shown that using normal distributions to represent such behavior leads to good predictions of application behavior [SB97].

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.

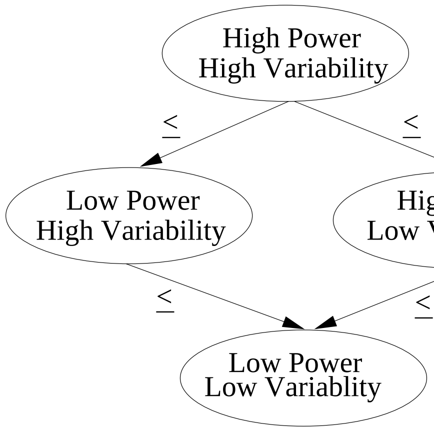

Figure 1. Diagram depicting the relative values of Tuning Factors for different configurations of power and variability.

We identify the partial ordering in Figure 1 for this requirement. 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 many other functions satisfy the relative ranking in Figure 1, and are also viable.

```
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($\mathbf{TP}_i$): A measure of the communication and computation 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($\mathbf{TP}_i$): 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*, *MidTuningFactor*, 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[3]. 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].

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.)

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

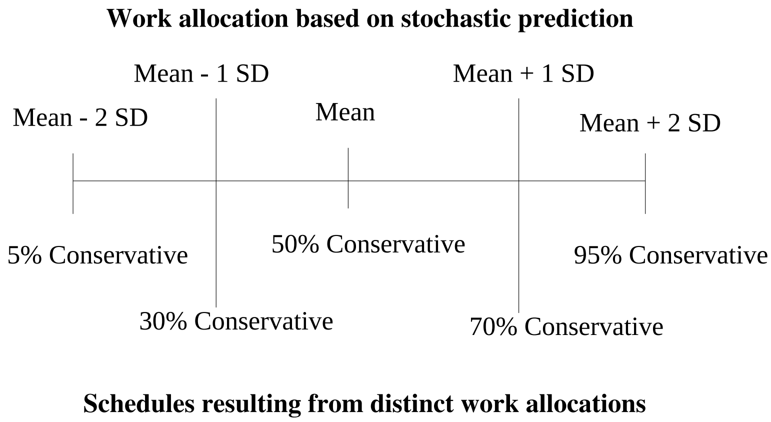

Figure 3. Diagram depicting the range of possible schedules given stochastic information.

In this extended abstract we focus only on SOR due to space constraints. Results for a distributed Genetic Algorithm and a distributed N-body code will be discussed in the final paper. 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 for a predefined number of iterations.

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.

3.1 Performance Metrics

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 two summary statistics. The first, 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.

The second summary statistic, 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

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.

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.

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

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

The third 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 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.

3.2 Experimental Data

Four representative experiments are shown in Figures 4 through 7. 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

Experiment	Mean	VTF	95TF
Figure 4	9	27	22
Figure 5	24	24	10
Figure 6	8	39	11
Figure 7	25	14	19

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

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 5	34.52	3.63	34.52	1.22	35.23	1.58
Figure 6	24.74	2.72	22.15	2.98	24.48	2.57
Figure 7	23.31	4.35	22.78	2.74	22.31	2.75

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

one had a medium variation and on the fourth had a high variation [4] Figure 5 shows a comparison of the three scheduling policies when there was a fairly constant low variance load on the Linux cluster. Figure 6 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 7 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.

4 Related Work

The stochastic scheduling work grew out of the adaptive scheduling focus of the AppLeS Project [BW96, BW97, BWF$^+$96].

Our approach to data allocation is based on a scheduling approach originally presented by Sih and Lee [SL90b, SL90a, SL93]. Their 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.

[4] 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.

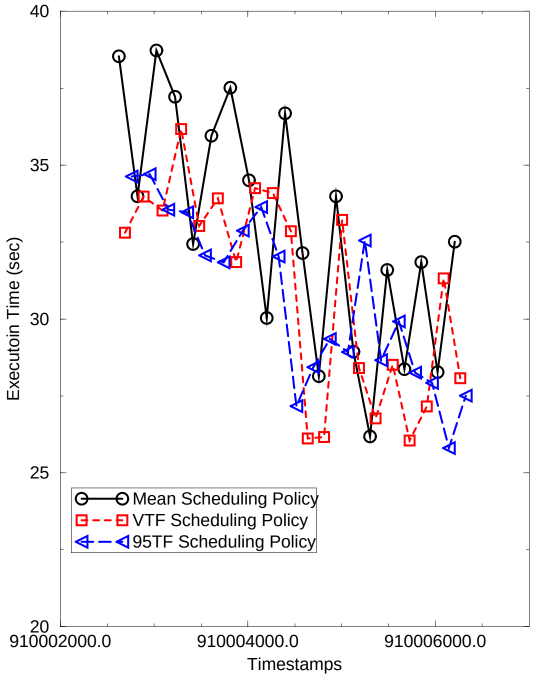

Figure 4. Comparison of Mean, VTF, and $95\mathrm{TF}$ policies, for the SOR benchmark on the Linux cluster with 2 low variability machines, one medium and one high.

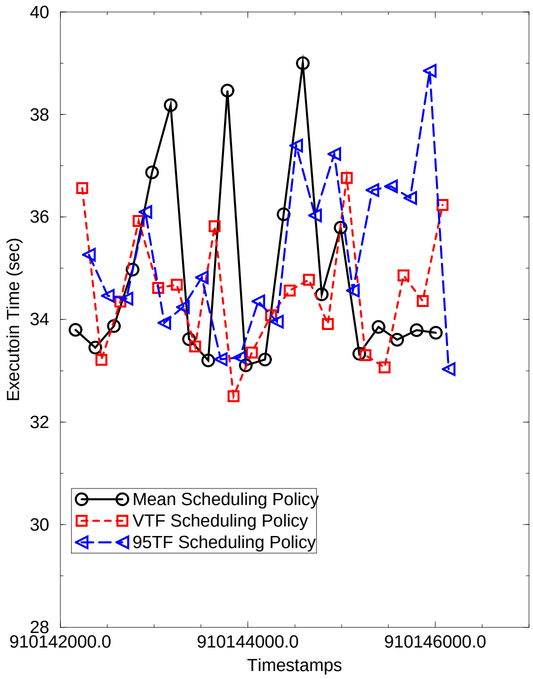

Figure 5. Comparison of Mean, VTF, and $95\mathrm{TF}$ policies, for the SOR benchmark on the Linux cluster for 4 low variability machines.

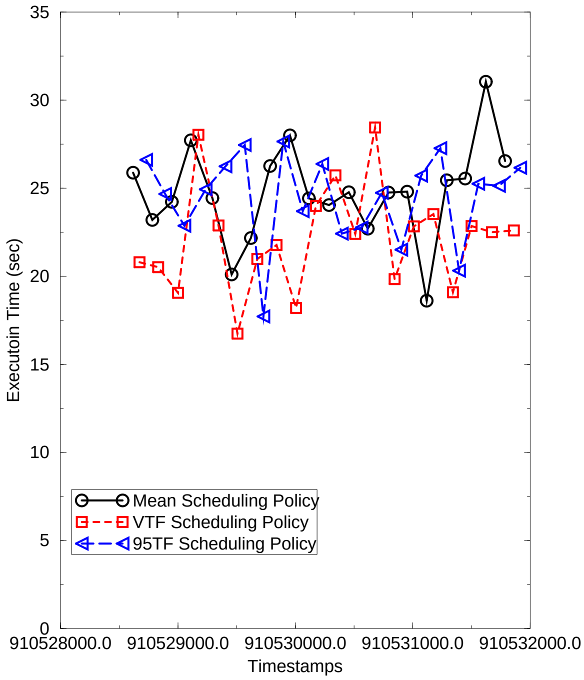

Figure 6. Comparison of Mean, VTF, and $\mathrm{95TF}$ policies, for the SOR benchmark on the PCL cluster when 2 of the machines had a low variability in available CPU and two had high.

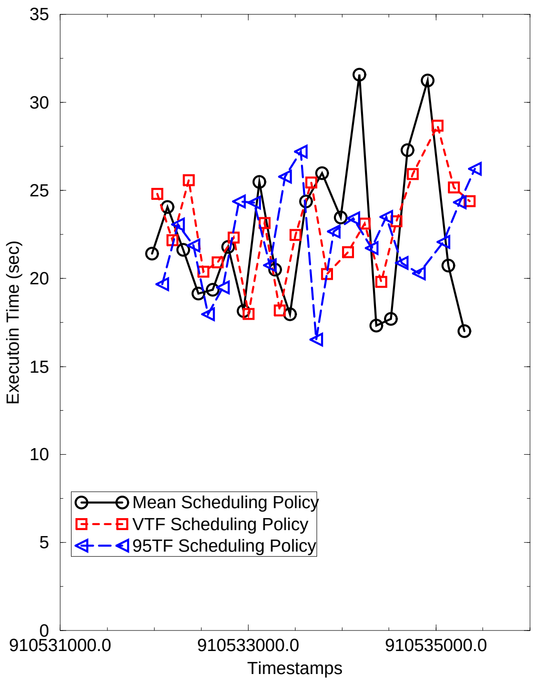

Figure 7. Comparison of Mean, VTF, and $\mathrm{95TF}$ policies, for the SOR benchmark on the PCL cluster when 1 of the machines had a low variability in available CPU and three had high.

Zaki et al. [ZLP96] also focused on the data allocation aspect of scheduling. Their 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.

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 apply this approach to the scheduling of a distributed SOR application. Our experimental results demonstrate that it is possible to obtain faster execution times and more predictable application behavior using stochastic scheduling.

We believe that adaptive techniques that can exploit variance and other representations of dynamic information will be critical to the development of successful techniques for application execution in distributed multi-user environments. The results reported here are promising in that they demonstrate that dynamic information can be used successfully to promote adaptation and improve application performance in these complex and challenging environments.

There are various possible areas for future work. The approach discussed here is one of many possible 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. Finally, similar extensions could be evaluated for other scheduling policies than time balancing as well.

Acknowledgements

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

References

[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.

[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.

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

[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`.

[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.

[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.

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