Chapter 1

USING PREDICTED VARIANCE FOR CONSERVATIVE SCHEDULING ON SHARED RESOURCES

Jennifer M. Schopf[1] and Lingyun Yang[2]

[1]*Mathematics and Computer Science Division, Argonne National Laboratory*
[2]*Computer Science Department, University of Chicago*

Abstract In heterogeneous and dynamic environments, efficient execution of parallel computations can require mappings of tasks to processors with performance that is both irregular and time varying. We propose a *conservative scheduling policy* that uses information about expected future variance in resource capabilities to produce more efficient data mapping decisions.

We first present two techniques to estimate future load and variance, one based on normal distributions and another using tendency-based prediction methodologies. We then present a family of stochastic scheduling algorithms that exploit such predictions when making data mapping decisions. We describe experiments in which we apply our techniques to an astrophysics application. The results of these experiments demonstrate that conservative scheduling can produce execution times that are significantly faster and less variable than other techniques.

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 is both irregular and time varying because of 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 selecting 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 that allows the scheduler to adapt application execution to the current system state, which means adapting to both the irregular (heterogeneous) nature of the resources and their time-varying behaviors.

We present a conservative scheduling technique that uses the predicted mean and variance of CPU capacity to make data mapping decisions. The basic idea is straightforward: We seek to allocate more work to systems that we expect to deliver the most computation, where this is defined from the viewpoint of the application. We often see that a resource with a larger capacity will also show a higher variance in performance and therefore will more strongly influence the execution time of an application than will a machine with less variance. Also, we keep in mind that a cluster may be homogeneous in machine type but quite heterogeneous in performance because of different underlying loads on the various resources.

Our conservative scheduling technique uses a conservative load prediction, equal to a prediction of the resource capacity over the future time interval of the application added to the predicted variance of the machine, in order to determine the proper data mapping, as opposed to just using a prediction of capacity as do many other approaches. This technique addresses both the dynamic and heterogeneous nature of shared resources.

We proceed in two steps. First, we define two techniques to predict future load and variance over a time interval, one based on using a normal distribution, the other using a tendency-based prediction technique defined in [YFS03]. Then, we use stochastic scheduling algorithms [SB99] that are parameterized by these predicted means and variances to make data distribution decisions. The result is an approach that exploits predicted variance in performance information to define a time-balancing scheduling strategy that improves application execution time.

We evaluate the effectiveness of this conservative scheduling technique by applying it to a particular class of applications, namely, loosely synchronous, iterative, data-parallel computations. Such applications are characterized by a single set of operations that is repeated many times, with a loose synchronization step between iterations [FJL+88, FWM94]. We present experiments conducted using Cactus [ABH+99, AAF+01], a loosely synchronous iterative computational astrophysics application. Our results demonstrate that we can achieve significant improvements in both mean execution time and the variance of those execution times over multiple runs in heterogeneous, dynamic environments.

2. RELATED WORK

Many researchers [Dai01, FB96, KDB02, WZ98] have explored the use of time balancing or load balancing models to reduce application execution time in heterogeneous environments. However, their work has typically assumed that resource performance is constant or slowly changing and thus does not take later variance into account. For example, Dail [Dai01] and Liu et al. [LYFA02] use the 10-second-ahead predicted CPU information provided by the Network Weather Service (NWS)([Wol98, WSH99a], also described in Chapter **??**) to guide scheduling decisions. While this one-step-ahead prediction at a time point is often a good estimate for the next 10 seconds, it is less effective in predicting the available CPU the application will encounter during a longer execution. Dinda et al. built a Running Time Advisor (RTA) [Din02] that predicts the running time of applications 1 to 30 seconds into the future based on a multistep-ahead CPU load prediction.

Dome [ABL$^+$95]i and Mars [GR96] support dynamic workload balancing through migration and make the application adaptive to the dynamic environment at runtime. But the implementation of such adaptive strategies can be complex and is not feasible for all applications.

In other work [SB99] we define the basic concept of stochastic values and their use in making scheduling decisions. This chapter extends that work to address the use of additional prediction techniques that originally predicted only one step ahead using a tendency-based approach [YFS03]. We define a time-balancing scheduling strategy based on a prediction of the next interval of time and a prediction of the variance (standard deviation) to counteract the problems seen with a one-step-ahead approach. Our technique achieves faster and less variable application execution time.

3. PROBLEM STATEMENT

Efficient execution in a distributed system can require, in the general case, mechanisms for the discovery of available resources, the selection of an application-appropriate subset of those resources, and the mapping of data or tasks onto selected resources. In this chapter we assume that the target set of resources is fixed, and we focus on the data mapping problem for data parallel applications.

We do not assume that the resources in this resource set have identical or even fixed capabilities in that they have identical underlying CPU loads. Within this context, our goal is to achieve data assignments that balance load between processors so that each processor finishes executing at roughly the same time, thereby minimizing execution time. This form of load balancing is also known as time balancing.

Time balancing is generally accomplished by solving a set of equations, such as the following, to determine the data assignments:

$$\begin{array}{l} E_i(D_i) = E_j(D_j) \ \ \forall\, i, j \\ \sum D_i = D_{Total}, \end{array} \tag{1.1}$$

where

- D_i is the amount of data assigned to processor i;
- D_{Total} is the total amount of data for the application;
- $E_i(D_i)$ is the execution time of task on processor i and is generally parameterized by the amount of data on that processor, D_i. It can be calculated by using a performance model of the application. For example, a simple application might have the following performance model:

$$\begin{array}{rl} E_i(D_i) = & Comm(D_i) * (futureNWCapacity) + \\ & Comp(D_i) * (futureCPUCapacity). \end{array} \tag{1.2}$$

Note that the performance of an application can be affected by the future capacity of both the network bandwidth behavior and the CPU availability.

In order to proceed, we need mechanisms for: (a) obtaining some measure of future capability and (b) translating this measure into an effective resource capability that is then used to guide data mapping. As we discuss below, two measures of future resource capability are important: the expected value and the expected variance in that value. One approach to obtaining these two measures is to negotiate a service level agreement (SLA) with the resource owner under which the owner would contract to provide the specified capability [CFK$^+$02]. Or, we can use observed historical data to generate a prediction for future behavior [Din02, SB99, SFT98, VS02, WSH99b, YFS03]. We focus in this chapter on the latter approach and present two techniques for predicting the future capability: using normal distributions and using a predicted aggregation. However, we emphasize that our results on topic (b) above are also applicable in the SLA-negotiation case.

4. PREDICTING LOAD AND VARIANCE

The Network Weather Service (NWS) [Wol98] provides predicted CPU information one measurement (generally about 10 seconds) ahead based on a time series of earlier CPU load information. Some previous scheduling work [Dai01, LYFA02] uses this one-step-ahead predicted CPU information as the future CPU capability in the performance model. For better data distribution and scheduling, however, what is really needed is an estimate of the average CPU load an application will experience during execution, rather than

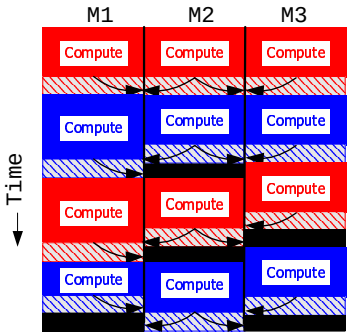

Figure 1.1. The interrelated influence among tasks of a synchronous iterative application.

the CPU information at a single future point in time. One measurement is simply not enough data for most applications.

In loosely synchronous iterative applications, tasks communicate between iterations, and the next iteration on a given resource cannot begin until the communication phase to that resource has been finished, as shown in Figure 1.1. Thus, a slower machine not only will take longer to run its own task but will also increase the execution time of the other tasks with which it communicates– and ultimately the execution time of the entire job. In Figure 1.1, the data was evenly divided among the resources, but M1 has a large variance in execution time. If M1 were running in isolation, it would complete the overall work in the same amount of time as M2 or M3. Because of its large variation, however,however, it is slow to communicate to M2 at the end of the second iteration, in turn delaying the task on M2 at the third computation step (in black), and hence delaying the task on M3 at the fourth computation step. Thus, the total job is delayed. It is this wave of delayed behavior caused by variance in the resource capability that we seek to avoid with our scheduling approach.

In the next subsections, we address two ways to more accurately predict longer-range load behavior: using a normal distribution and extending a one-step-ahead load prediction developed in previous work [YFS03]

4.1 Normal Distribution Predictions

Performance models are often parameterized by values that represent system 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 way—as distributions [SB98], as intervals [SB99], and as histograms [Sch99]. In this chapter we assume that we can adequately represent stochastic values using normal distributions. Normal distributions, also called Gaussian distributions, are representative of large collections of random variables. As such, many real phenomena in computer systems generate distributions that are close to normal distributions [Adv93, AV93].

A normal distribution can be defined by the formula

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2}, \qquad -\infty < x < \infty \tag{1.3}$$

for parameters μ, the *mean*, which gives the *center* of the range of the distribution, and σ, the *standard deviation*, which describes the variability in the distribution and gives a range around the mean. Normal distributions are symmetric and bell shaped and have the property that the range defined by the mean plus and minus two standard deviations captures approximately 95% of the values of the distribution.

Figure 1.2 shows a histogram of runtimes for an SOR benchmark on a single workstation with no other users present, and the normal distribution based on the data mean, m, and standard deviation, sd. Distributions can be represented graphically in two common ways: by the *probability density function* (PDF), as shown on the left in Figure 1.2, which graphs values against their probabilities, similar to a histogram, and by the *cumulative distribution function* (CDF), as shown on the right in Figure 1.2, which illustrates the probability that a point in the range is less than or equal to a particular value.

In the following subsections we describe the necessary compositional arithmetic to use normal distributions in predictive models; in Sections 4.1.2 and 4.1.3 we discuss alternatives to consider when the assumption of a normal distribution is too far from the actual distribution of the stochastic value.

4.1.1 Arithmetic Operations over Normal Distributions

In order for prediction models to use stochastic values, we need to provide a way to combine stochastic values arithmetically. In this subsection we define common arithmetic interaction operators for stochastic values represented by normal distributions by taking advantage of the fact that normal distributions are closed under linear combinations [LM86].

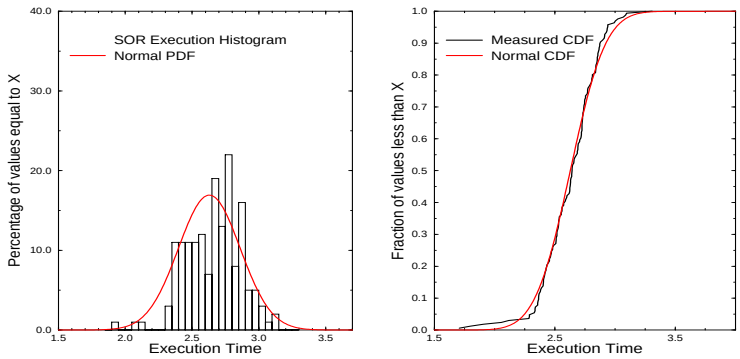

Figure 1.2. Graphs showing the PDF and CDF of SOR benchmark with normal distribution based on data mean and standard deviation.

For each arithmetic operation, we define a rule for combining stochastic values based on standard statistical error propagation methods [Bar78]. In the following, we assume that point values are represented by P and all stochastic values are of the form (m_i, sd_i) and represent normal distributions, where m_i is the mean and sd_i is the standard deviation.

When combining two stochastic values, two cases must be considered: correlated and uncorrelated distributions. Two distributions are *correlated* when there is an association between them, that is, they jointly vary in a similar manner [DP96a]. More formally, correlation is the degree to which two or more attributes, or measurements, on the same group of elements show a tendency to vary together. For example, when network traffic is heavy, available bandwidth tends to be low, and latency tends to be high. When network traffic is light, available bandwidth tends to be high, and latency tends to be low. We say that the distributions of latency and bandwidth are correlated in this case.

When two stochastic values are *uncorrelated*, they do not jointly vary in a similar manner. This case may occur when the time between measurements of a single quantity is large or when the two stochastic values represent distinct characteristics. For example, available CPU on two machines not running any applications in common may be uncorrelated.

Table 1.1 summarizes the arithmetic operations between a stochastic value and a point value, two stochastic values from correlated distributions, and two stochastic values from uncorrelated distributions.

Note that the product of stochastic values with normal distributions does not itself have a normal distribution. Rather, it is long-tailed. In many circumstances, we can approximate the long-tailed distribution with a normal distribution and ignore the tail, as discussed below in Section 4.1.2.

Table 1.1. Arithmetic combinations of a stochastic value with a point value and with other stochastic values [Bar78].

	Addition	Multiplication
Point Value and Stochastic Value	$(m_i, sd_i) + P = ((m_i + P), sd_i)$	$P(m_i, sd_i) = (Pm_i, Psd_i)$
Stochastic Values with Correlated Distributions	$\sum_{i=1}^{n}(m_i, sd_i) = \left(\sum_{i=1}^{n} m_i, \quad \sum_{i=1}^{n} \lvert sd_i \rvert\right)$	$(m_i, sd_i)(m_j, sd_j) = (m_i m_j, (sd_i m_j + sd_j m_i + sd_i sd_j))$
Stochastic Values with Uncorrelated Distributions	$\sum_{i=1}^{n}(m_i, sd_i) \approx \left(\sum_{i=1}^{n} m_i, \sqrt{\sum_{i=1}^{n} sd_i^2}\right)$	$(m_i, sd_i)(m_j, sd_j) \approx \left(m_i m_j, \left(m_i m_j \sqrt{\left(\frac{sd_i}{m_i}\right)^2 + \left(\frac{sd_j}{m_j}\right)^2}\right)\right)$

4.1.2 Using Normal Distributions to Represent Nonnormal Stochastic Model Parameters

In this section, we provide examples of stochastic parameters that are not normal but can often be adequately represented by normal distributions.

Not all system characteristics can be accurately represented by normal distributions. Figure 1.3 shows the PDF and CDF for bandwidth data between two workstations over 10 Mbit Ethernet. This is a typical graph of a *long-tailed distribution*; that is, the data has a threshold value and varies monotonically from that point, generally with the median value several points below (or above) the threshold. A similarly shaped distribution, shown in Figure 1.4 on the left, may be found in data resulting from dedicated runs of a nondeterministic distributed genetic algorithm code.

Neither of these distributions is normal; however, it may be adequate to approximate them by using normal distributions. Normal distributions are a good substitution for long-tailed model parameters only when inaccuracy in the predictions generated by the structural model can be tolerated by the scheduler, performance model, or other mechanism that uses the data.

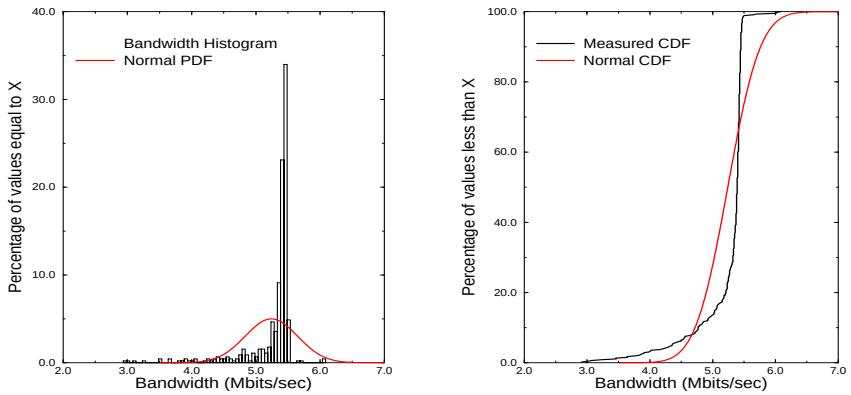

Figure 1.3. Graphs showing the PDF and CDF for bandwidth between two workstations over 10 Mbit Ethernet with long-tailed distribution and corresponding normal distribution.

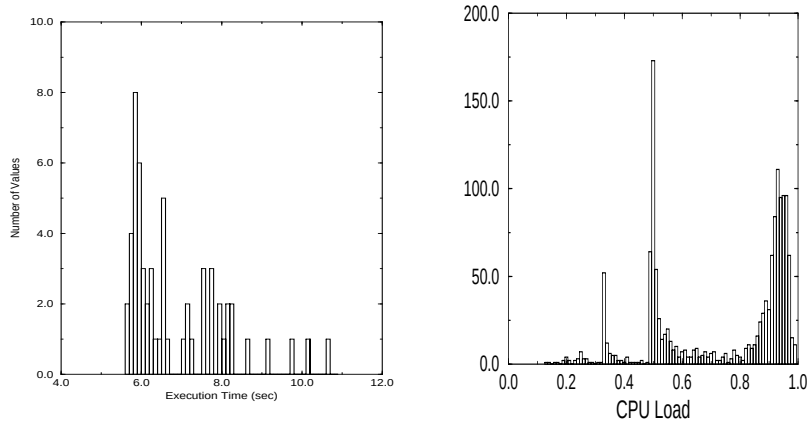

Figure 1.4. Two examples of nonnormal distribution behavior: a histogram of a nondeterministic application on the left; Available CPU on a production workstation on the right.

Alternatively, some model parameters are best represented by multimodal distributions. One characterization of the general shape of a distribution is the number of peaks, or *modes*. A distribution is said to be *unimodal* if it has a single peak, *bimodal* if it has two peaks, and *multimodal* if it has more than two peaks. Figure 1.4 on the left shows a histogram of available CPU data for an Ultra Sparc workstation running Solaris taken over 12 hours using `vmstat`. The Unix tool `vmstat` reports the exact CPU activity at a given time, in terms of the processes in the run queue, the blocked processes, and the swapped processes as a snapshot of the system every n seconds (where for our trace, n

= 5). For this distribution, the majority of the data lies in three modes: a mode centered at 0.94, a mode centered at 0.49, and a mode centered at 0.33.

For this data, the modes are most likely an artifact of the scheduling algorithm of the operating system. Most Unix-based operating systems use a round-robin algorithm to schedule CPU bound processes: When a single process is running, it receives all of the CPU; when two processes are running, each uses approximately half of the CPU; when there are three, each gets a third; and so forth. This is the phenomenon exhibited in Figure 1.4.

To represent a modal parameter using a normal distribution, we need to know whether values represented by the parameter remain within a single mode during the timeframe of interest. If the values of the parameter remain within a single mode (i.e. if they exhibit *temporal locality*), we can approximate the available CPU as a normal distribution based on the data mean and standard deviation of the appropriate mode without excessive loss of information. An example of this (from a 24-hour trace of CPU loads) is shown as a time series in Figure 1.5 on the left.

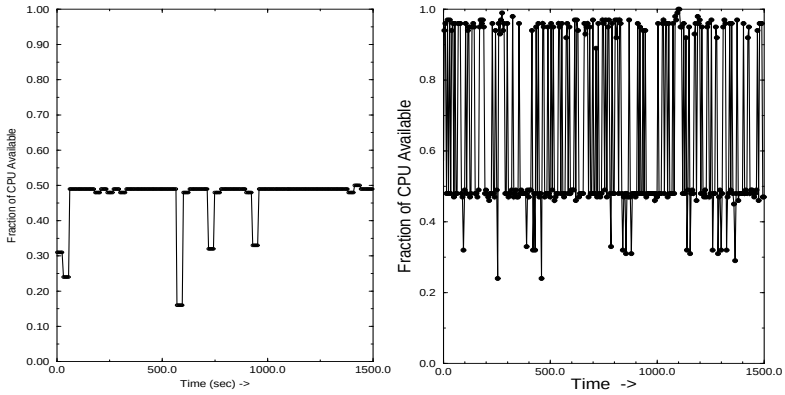

Figure 1.5. Two time series showing temporal locality (on the left) and nontemporal locality (on the right) for CPU data.

If the values of the parameter change modes frequently or unpredictably, we say that that the data exhibits *temporal nonlocality*. An example of this, taken from the same 24-hour CPU trace as before, is shown as a time series in Figure 1.5 on the right. In this case, some way of deriving a prediction must be devised that takes into account the fluctuation of the parameter data between multiple modes.

A brute-force approach to representing multimodal data would be to simply ignore the multimodality of the data and represent the stochastic value as a normal distribution based on the mean and standard deviation of the data as a whole. This approximation is, however, unlikely to capture the relevant behavior characteristics of the data with any accuracy. Because of the multi-

modal behavior, a mode with a small variance in actuality may end up being represented by a normal distribution with a large variance (and a large standard deviation). If this brute-force method were used for the data in Figure 1.5, the mean would be 0.66 and the standard deviation would be 0.24.

An alternative approach is to calculate an *aggregate mean* and *aggregate standard deviation* for the value based on the mean and standard deviation for each mode. Let (m_i, sd_i) represent the mean and the standard deviation for the data in mode i. We define the *aggregate mean (AM)* and the *aggregate standard deviation (ASD)* of a multimodal distribution by

$$AM = \sum p_i(m_i) \tag{1.4}$$

$$ASD = \sum p_i(sd_i), \tag{1.5}$$

where p_i is the percentage of data in mode i. Since we represent each individual mode in term of a normal distribution, (AM, ASD) will also have a normal distribution. For the data in Figure 1.5, AM = 0.68 and ASD = 0.031.

Note that using the aggregate mean and the aggregate standard deviation is an attempt to define a normal distribution that is somehow close to the multimodal distribution. Determining whether two distributions are close is itself an interesting problem that we discuss briefly in the subsection below.

4.1.3 When Is a Distribution *Close* to Normal?

In the preceding subsections, we made a key assumption that the values in the distribution were *close* to (could be adequately represented by) normal distributions. To define "close," we can consider several methods for determining the similarity between a given data set and the normal distribution represented by its data mean and its data standard deviation.

One common measurement of goodness of fit is the chi-squared (χ^2) technique [DP96b]. This is a quantitative measure of the extent to which observed counts differ from the expected counts over a given range, called a cell. The value for χ^2 is the sum of a goodness of fit for all quantities

$$\chi^2 = \sum_{all\ cells} \frac{(\text{observed cell count - expected cell count})^2}{\text{expected cell count}} \tag{1.6}$$

for the observed data and the expected data resulting from the normal distribution. The value of the χ^2 statistic reflects the magnitude of the discrepancies between observed and expected cell counts: a larger value indicates larger discrepancies.

Another metric of closeness in the literature is called 1-distance between PDF's [MLH95], where

$$\|f_n - f_d\|_1 = \int_{-\infty}^{\infty} |f_n(x) - f_p(x)| dx \tag{1.7}$$

for function f_d for the data and f_n for the normal distribution based on the data mean and standard deviation. This corresponds to a maximal error between the functions.

For both of these metrics, a user or scheduler would need to determine a threshold for closeness acceptable for their purposes.

If we approximate nonnormal data using a normal distribution, there may be several effects. When the distribution of a stochastic value is represented by a normal distribution but is not actually normal, arithmetic operations might exclude values that they should not. By performing arithmetic on the mean and standard deviation, we are able to use optimistic formulas for uncorrelated values in order to narrow the range that is considered in the final prediction. If the distributions of stochastic values were actually long-tailed, for example, this might cut off values from the tail in an unacceptable way.

Normal distributions are closed under linear combinations [LM86], but general distributions are not. If we use arithmetic rules defined for normal distributions on nonnormal data, we have no information about the distribution of the result. Further, it may not be possible to ascertain the distribution of a stochastic value, or the distribution may not be sufficiently close to normal. In such cases, other representations must be used. In the next section, we explore an alternative for representing stochastic values using an aggregated prediction technique.

4.2 Aggregate Predictions

In this section we describe how a time series predictor can be extended to obtain three types of predicted CPU load information: the next step predicted CPU load at a future time point (Section 4.2.1); the average interval CPU load for some future time interval (Section 4.2.2); and the variation of CPU load over some future time interval (Section 4.2.3).

4.2.1 One-Step-Ahead CPU Load Prediction

The tendency-based time series predictor developed in our previous work can provide one-step-ahead CPU load prediction based on history CPU load

```
// Determine Tendency
      if ((V_(T-1) - V_T )<0)
 Tendency="Increase";
    else if ((V_T - V_(T-1)))<0)
           Tendency="Decrease";
      if (Tendency="Increase") then
             PT+1 = V_T + IncrementConstant;
             IncrementConstant adaptation process
      else if (Tendency="Decrease") then
             PT+1 = V_T - V_T*DecrementFactor;
             DecrementFactor adaptation process
```

Figure 1.6. Psuedo-code for Tendency algorithm.

information [YFS03]. This predictor has been demonstrated to be more accurate than other predictors for CPU load data. It achieves prediction errors that are between 2% and 55% less (36% less on average) than those incurred by the predictors used within the NWS on a set of 38 machines load traces. The algorithm predicts the next value according to the tendency of the time series change assuming that if the current value increases, the next value will also increase and that if the current value decreases, the next value will also decrease.

Given the preceding history data measured at a constant-width time interval, our mixed tendency-based time series predictor uses the algorithm in Figure 1.6, where V_T is the measured value at the Tth measurement and $PT + 1$ is the predicted value for measurement value V_{T+1}.

We find that a mixed-variation (that is, different behavior for the increment from that of the decrement) experimentally performed best. The IncrementConstant is set initially to 0.1, and the DecrementFactor is set to 0.01. At each time step, we measure the real data (V_{T+1}) and calculate the difference between the current measured value and the last measured value (V_T) to determine the real increment (decrement) we should have used in the last prediction in order to get the actual value. We adapt the value of the increment (decrement) value accordingly and use the adapted IncrementConstant (or DecrementFactor) to predict the next data point.

Using this time series predictor to predict the CPU load in the next step, we treat the measured preceding CPU load time series as the input to the predictor. The predictor's output is the predicted CPU load at the next step, P_{n+1}. So if the time series $C = c_1, c_2 \ldots c_n$ is the CPU load time series measured at constant-width time interval and is used as input to the predictor, the result is the predicted value P_{n+1} for the measurement value c_{n+1}.

4.2.2 Interval Load Prediction

Instead of predicting one step ahead, we want to be able to predict the CPU load over the time interval during which an application will run. Since the CPU load time series exhibits a high degree of self-similarity [Din99], averaging values over successively larger time scales will not produce time series that are dramatically smoother. Thus, to calculate the predicted average CPU load an application will encounter during its execution, we need to first aggregate the original CPU load time series into an interval CPU load time series, then run predictors on this new interval time series to estimate its future value.

Aggregation, as defined here, consists of converting the original CPU load time series into an interval CPU load time series by combining successive data over a nonoverlapping larger time scale. The aggregation degree, M, is the number of original data points used to calculate the average value over the time interval. This value is determined by the resolution of the original time series and the execution time of the applications, and need be only approximate.

For example, the resolution of the original time series is 0.1 Hz, or measured every 10 seconds, and if the estimated application execution time is about 100 seconds, the aggregation degree M can be calculated by

$$\begin{aligned} M &= ExecTimeOfApplication * FreqOfOriginalTimeSeries \\ &= 100 * 0.1 \\ &= 10 \end{aligned} \tag{1.8}$$

Hence, the aggregation degree is 10. In other words, 10 data points from the original time series are needed to calculate one aggregated value over 100 seconds. The process of aggregation consists of translating the incoming time series, ($C = c_1, c_2, \ldots c_n$), into the aggregated time series, ($A = a_1, a_2, \ldots a_k$), such that

$$a_i = \frac{\sum_{j=1..M} C_{n-(k-i+1)*M+j}}{M} \tag{1.9}$$

for $i = 1 \ldots k$i for $k = \lceil \frac{n}{M} \rceil$. Each value in the interval CPU load time series a_i is the average CPU load over the time interval that is approximately equal to the application execution time.

After the aggregated time series is created, the second step of our interval load prediction involves using the one-step-ahead predictor on the aggregated time series to predict the mean interval CPU load. So the aggregated time series A_i is fed into the one-step-ahead predictor, resulting in pa_{K+1}, the predicted value of a_{k+1}, which is approximately equal to the average CPU load the application will encounter during execution.

4.2.3 Load Variance Prediction

To predict the variation of CPU load, for which we use standard deviation, during the execution of an application, we need to calculate the standard deviation time series using the original CPU load time series C and the interval CPU load time series A (defined in the preceding section).

Assuming the original CPU load time series is $C = c_1, c_2, \ldots c_n$, the interval load time series is $A = a_1, a_2, \ldots a_k$, and an aggregation degree of M, we can calculate the standard deviation CPU load time series $S = s_1, s_2, \ldots s_k$:

$$S_i = \sqrt{\sum_{j=1\ldots M} \frac{\left(C_{n-(k-i+1)*M+j-A_i}\right)^2}{M}} \tag{1.10}$$

for $i = 1 \ldots k$.

Each value in standard deviation time series s_i is the average difference between the CPU load and the mean CPU load over the interval.

To predict the standard deviation of the CPU load, we use the one-step-ahead predictor on the standard deviation time series. The output ps_{k+1} will be the predicted value of s_{k+1}, or the predicted CPU load variation for the next time interval.

5. APPLICATION SCHEDULING

Our goal is to improve data mapping in order to reduce total application execution time despite resource contention. To this end, we use the time-balancing scheduling algorithm described in Section 3, parameterized with an estimate of future resource capability.

5.1 Cactus Application

We apply our scheduling algorithms in the context of Cactus, a simulation of a 3D scalar field produced by two orbiting astrophysical sources. The solution is found by finite differencing a hyperbolic partial differential equation for the scalar field. This application decomposes the 3D scalar field over processors and places an overlap region on each processor. For each time step, each processor updates its local grid point and then synchronizes the boundary values. It is an iterative, loosely synchronous application, as described in Section 4. We use a one-dimensional decomposition to partition the workload in our experiments. The full performance model for Cactus is described in [LYFA02], but in summary it is

$$\begin{aligned} E_i(D_i) = \quad & startUpTime + (D_i * Comp_i + Comm_i) \\ & *slowdown(\textit{effective CPU load}) \end{aligned} \tag{1.11}$$

$Comp_i$ and $Comm_i$, the computation time of per data point and communication time of the Cactus application in the absence of contention, can be calculated by formulas described in [RIF01]. We incur a startup time when initiating computation on multiple processors in a workstation cluster that was experimentally measured and fixed. The function *slowdown(effective CPU load)*, which represents the contention effect on the execution time of the application, can be calculated by using the formula described in [LYFA02].

The performance of the application is greatly influenced by the actual CPU performance achieved in the presence of contention from other competing applications. The communication time is less significant when running on a local area network, but for wide-area network experiments this factor would also be parameterized by a capacity measure.

Thus, our problem is to determine the value of CPU load to be used to evaluate the slowdown caused by contention. We call this value the effective CPU load and equate it to the average CPU load the application will experience during its execution.

5.2 Scheduling Approaches

As shown in Figure 1, variations in CPU load during task execution can also influence the execution time of the job because of interrelationships among tasks. We define a conservative scheduling technique that always allocates less work to highly varying machines. For the purpose of comparison, we define the effective CPU load in a variety of ways, each giving us a slightly different scheduling policy. We define five policies to compare in the experimental section:

- One-step scheduling (OSS): Use the one-step-ahead prediction of the CPU load, as described in Sections 4.2.1, for the effective CPU load.
- Predicted mean interval scheduling (PMIS): Use the interval load prediction, described in Section 4.2.2, for the effective CPU load.
- Conservative scheduling (CS): Use the conservative load prediction, equal to the interval load prediction (defined in Section 4.2.2) added to a measure of the predicted variance (defined in Section 4.2.3) for the effective CPU load. That is, effective CPU load= $pa_{k+1} + ps_{k+1}$.
- History mean scheduling (HMS): Use the mean of the history CPU load for the 5 minutes preceding the application start time for the value for effective CPU load. This approximates the estimates used in several common scheduling approaches [TSC00, WZ98].
- History conservative scheduling (HCS): Use the conservative estimate CPU load defined by using the normal distribution stochastic value de-

fined in Section 4.1. In practice, this works out to adding the mean and variance of the history CPU load collected for 5 minutes preceding the application run as the effective CPU load.

6. EXPERIMENTS

To validate our work, we conducted experiments on workstation clusters at the University of Illinois at Champaign-Urbana (UIUC) and the University of California, San Diego (UCSD), which are part of the GrADS testbed [BCC+01]

6.1 Experimental Methodology

We compared the execution times of the Cactus application with the five scheduling policies described in Section 5: one-step scheduling (OSS), predicted mean interval scheduling (PMIS), conservative scheduling (CS), history mean scheduling (HMS), and history conservative scheduling (HCS).

At UIUC, we used a cluster of four Linux machines, each with a 450 MHz CPU; at UCSD, we used a cluster of six Linux machines, four machines with a 1733 MHz CPU, one with a 700 MHz CPU, and one with a 705 MHz CPU. All machines were dedicated during experiments.

To evaluate the different scheduling polices under identical workloads, we used a load trace playback tool [DO00] to generate a background workload from a trace of the CPU load that results in realistic and repeatable CPU contention behavior. We chose nine load time series available from [Yan03]. These are all traces of actual machines, which we characterize by their mean and standard deviation. We used 100 minutes of each trace, at a granularity of 0.1 Hz. The statistic properties of these CPU load traces are shown in Table 1.2. Note that even though some machines have the same speed, the performance that they deliver to the application varied that they each experienced different background loads.

Table 1.2. The mean and standard deviation of 9 CPU load traces.

CPU Load Trace Name	Machine Name	Mean	SD
LL1	abyss	0.12 (L)	0.16 (L)
LL2	axp7	0.02 (L)	0.06 (L)
LH1	vatos	0.22 (L)	0.31 (H)
LH2	axp1	0.14 (L)	0.29 (H)
HL1	mystere	1.85 (H)	0.14 (L)
HL2	pitcairn	1.19 (H)	0.12 (L)
HH	axp0	1.07 (H)	0.48 (H)
HH2	axp10	1.18 (H)	0.31 (H)

6.2 Experimental Results

Results from four representative experiments are shown in Figures 1.7–1.10. A summary of the testbeds and the CPU load traces used for the experiments is given in Table 1.3.

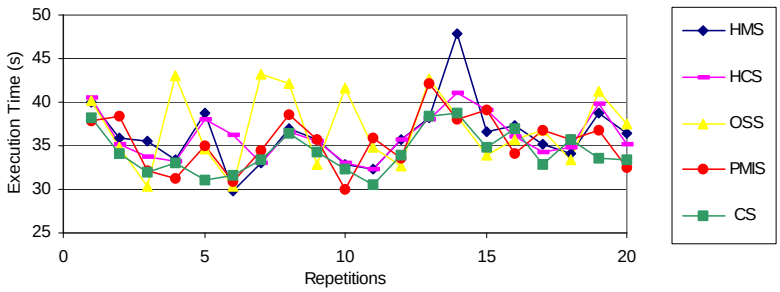

Figure 1.7. Comparison of the history mean, history conservative, one-step, predicted mean interval and conservative scheduling policies on the UIUC cluster with two low-variance machines (one with a low mean and the other with a high mean) and two high-variance machines (one with a low mean, the other with a high mean).

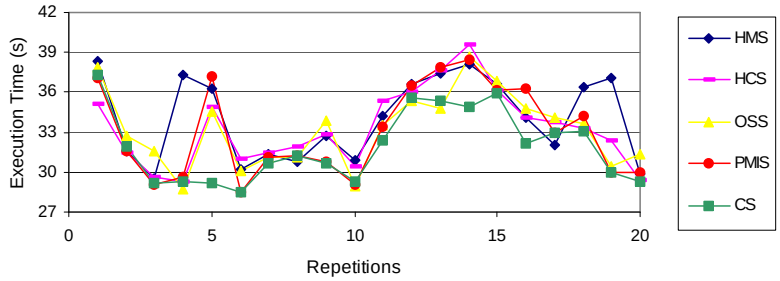

Figure 1.8. Comparison of the history mean, history conservative, one-step, predicted mean interval and conservative scheduling policies on the UIUC cluster with two low-variance machines and two high-variance machines (all with a low mean).

Table 1.3. CPU load traces used for each experiment.

Experiments	Testbed	CPU Load Traces
Fig. 1.7	UIUC	LL1, LH1, HL1, HH1
Fig. 1.8	UIUC	LL1, LL2, LH1, LH2
Fig. 1.9	UCSD	LL1, LL2, LH1, LH2, HL1, HL2
Fig. 1.10	UCSD	LH1, LH2, HL1, HL2, HH1, HH2

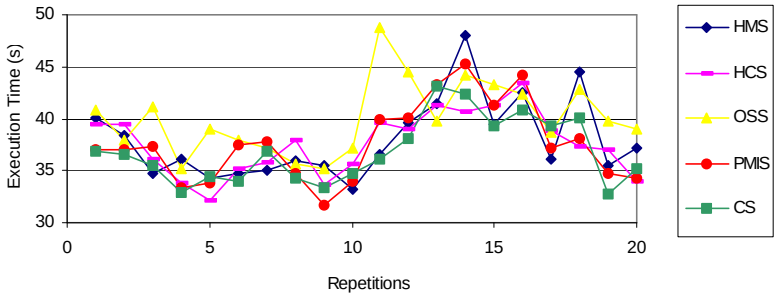

Figure 1.9. Comparison of the history mean, history conservative, one-step, predicted mean interval and conservative scheduling policies on the UCSD cluster with four low-variance machines (one with a low means and two with a high means) and two high-variance machines (with low means).

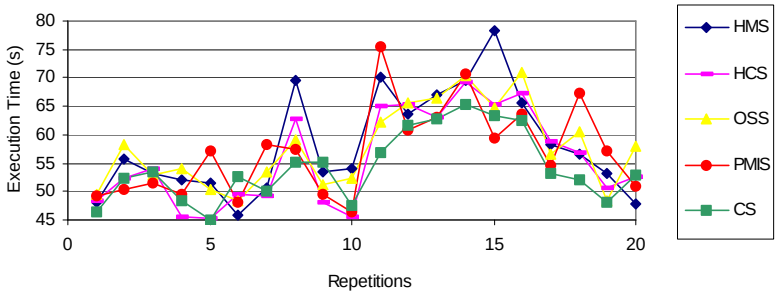

Figure 1.10. Comparison of the history mean, history conservative, one-step, predicted mean interval and conservative scheduling policies on the UCSD cluster with two low-variance machines (all with high means) and four high-variance machines (two with a low mean, two with a high mean).

To compare these policies, we used two metrics: an absolute comparison of run times and a relative measure of achievement. The first metric involves an average mean and an average standard deviation for the set of runtimes of each scheduling policy as a whole, as shown in Table 1.4. This metric gives a rough valuation on the performance of each scheduling policy over a given interval of time. Over the entire run, the conservative scheduling policy exhibited 2%–7% less overall execution time than history mean and history conservative scheduling policies, by using better information prediction, and 1.2%–7% less overall execution time than did the one-step and predicted mean interval scheduling policies. We also see that taking variation information into account in the scheduling policy results in more *predictable* application behavior: The history conservative scheduling policy exhibited 9%–29% less standard deviation of execution time than did the history mean. The conservative scheduling

Table 1.4. Average mean and average standard deviation for entire set of runs for each scheduling policy.

Exp.	HMS		HCS		OSS		PMIS		CS	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD
Fig. 1.7	36.2	3.7	36.1	2. 6	37.0	4.2	35.4	3.2	34.3	2.4
Fig. 1.8	34.1	3.1	33.3	2.8	33.2	2.7	33.0	3.4	31.9	2.7
Fig. 1.9	38.0	3.8	37.6	3.0	37.8	3.5	37.6	3.8	36.8	3.1
Fig. 1.10	58.2	9.1	55.7	8.1	57.7	7.2	57.0	8.0	54.2	6.1

policy exhibited 1.5%–41% less standard deviation in execution time than the one-step scheduling policy and 20%–41% less standard deviation of execution time than the predicted mean interval scheduling policy.

The second metric we used, *Compare*, is a relative metric that evaluates how often each run achieves a minimal execution time. We consider a scheduling policy to be better than others if it exhibits a lower execution time than another policy on a given run. Five possibilities exist: *best* (best execution time among the five policies), *good* (better than three policies but worse than one), *average* (better than two policies and worse than two), *poor* (better than one policy but worse than three), and *worst* (worst execution time of all five policies).

These results are given in Table 1.5, with the largest value in each case shown in boldface. The results indicate that conservative scheduling using predicted mean and variation information is more likely to have a *best* or *good* execution time than the other approached on both clusters. This fact indicates that taking account of the average and variation CPU information during the period of application running in the scheduling policy can significantly improve the application's performance.

To summarize our results: independent of the loads and CPU capabilities considered on our testbed, the conservative scheduling policy based on our tendency-based prediction strategy with mixed variation achieved better results than the other policies considered. It was both the best policy in more situations under all load conditions on both clusters, and the policy that resulted in the shortest execution time and the smallest variation in execution time.

7. CONCLUSIONS AND FUTURE WORK

We have proposed a conservative scheduling policy able to achieve efficient execution of data-parallel applications even in heterogeneous and dynamic environments. This policy uses information about the expected mean and variance of future resource capabilities to define data mappings appropriate for dynamic resources. Intuitively, the use of variance information is appealing

Table 1.5. Summary statistics using Compare to evaluate five scheduling policies.

Experiment	Policy	Best	Good	Avg	Poor	Worst
Fig. 1.7	HMS	2	2	7	3	6
	HCS	1	4	6	6	3
	OSS	5	5	0	2	8
	PMIS	6	2	3	7	2
	CS	6	7	4	2	1
Fig. 1.8	HMS	2	2	5	5	6
	HCS	2	3	4	6	5
	OSS	4	2	5	3	6
	PMIS	1	8	3	5	3
	CS	11	5	3	1	0
Fig. 1.9	HMS	4	3	5	4	4
	HCS	4	3	7	4	2
	OSS	1	1	4	4	10
	PMIS	1	10	0	6	3
	CS	10	3	4	2	1
Fig. 1.10	HMS	2	2	5	7	4
	HCS	4	3	6	5	2
	OSS	0	3	6	5	6
	PMIS	4	8	1	1	6
	CS	10	4	2	2	2

because it provides a measure of resource reliability. Our results suggest that this intuition is valid.

Our work comprises two distinct components. First, we show how to obtain predictions of expected mean and variance information. Then we show how information about expected future mean and variance (as obtained, for example, from our predictions) can be used to guide data mapping decisions. In brief, we assign less work to less reliable (higher variance) resources, thus protecting ourselves against the larger contending load spikes that we can expect on those systems. We apply our prediction techniques and scheduling policy to a substantial astrophysics application. Our results demonstrate that our techniques can obtain better execution times and more predictable application behavior than previous methods that focused on predicted means alone. While the performance improvements obtained are modest, they are obtained consistently and with no modifications to the application beyond those required to support nonuniform data distributions.

We are interested in extending this work to other dynamic system information, such as network status. Another direction for further study is a more sophisticated scheduling policy that may better suit other particular environments and applications.

Acknowledgments

We are grateful to Peter Dinda for permitting us to use his load trace play tool, and to our colleagues within the GrADS project for providing access to testbed resources. This work was supported in part by the Grid Application Development Software (GrADS) project of the NSF Next Generation Software program, under Grant No. 9975020, and in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under contract W-31-109-Eng-38.

References

[AAF+01] G. Allen, D. Angulo, I. Foster, G. Lanfermann, C. Liu, T. Radke, E. Seidel, and J. Shalf. The Cactus worm: Experiments with dynamic resource discovery and allocation in a Grid environment. *International Journal of High Performance Computing Applications*, 15(4):345–358, 2001.

[ABH+99] G. Allen, W. Benger, C. Hege, J. Masso, A. Merzky, T. Radke, E. Seidel, and J. Shalf. Solving Einstein's equations on supercomputers. *IEEE Computer Applications*, 32(12):52–58, 1999.

[ABL+95] Jose Nagib Cotrim Arabe, Adam Beguelin, Bruce Lowekamp, Erik Seligman, Mike Starkey, and Peter Stephan. Dome: Parallel programming in a heterogeneous multi-user environments. Technical Report CMU-CS-95-137, Carnegie Mellon University, School of Computer Science, 1995.

[Adv93] Vikram S. Adve. *Analyzing the Behavior and Performance of Parallel Programs*. PhD thesis, University of Wisconsin-Madison, December 1993. Also available as University of Wisconsin Computer Sciences Technical Report #1201.

[AV93] Vikram Adve and Mary Vernon. The influence of random delays on parallel execution times. In *Proceedings of Sigmetrics '93*, 1993.

[Bar78] B. Austin Barry. *Errors in Practical Measurement in Science, Engineering and Technology*. John Wiley & Sons, 1978.

[BCC+01] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, D. Reed, L. Torczon, and R. Wolski. The GrADS project: Software support for high-level Grid application development. *International Journal of High-Performance Computing Applications*, 15(4):327–344, 2001.

[CFK+02] K. Czajkowski, I. Foster, C. Kesselman., V. Sander, and S. Tuecke. SNAP: A protocol for negotiating service level agreements and coordinating resource management in distributed systems. In D. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing (Proceedings of the Eighth International JSSPP Workshop; LNCS #2537)*, pages 153–183. Springer-Verlag, 2002.

[Dai01] H. J. Dail. A modular framework for adaptive scheduling in Grid application development environments. Technical Report CS2002-0698, Computer Science Department, University of California, California, San Diego, 2001.

[Din99] P. A. Dinda. The statistical properties of host load. *Scientific Programming*, 7:3–4, Fall 1999.

[Din02] P. A. Dinda. Online prediction of the running time of tasks. *Cluster Computing*, 5(3), 2002.

[DO00] P. A. Dinda and D. R. O'Hallaron. Realistic CPU workloads through host load trace playback. In *Proceedings of the Fifth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR 2000)*, 2000.

[DP96a] Jay Devore and Roxy Peck. *Statistics: The Exploration and Analysis of Data*, page 88. Duxbury Press, 1996.

[DP96b] Jay Devore and Roxy Peck. *Statistics: The Exploration and Analysis of Data*, page 567. Duxbury Press, 1996.

[FB96] S.M. Figueira and F. Berman. Mapping parallel applications to distributed heterogeneous systems. Technical Report UCSD CS Tech Report # CS96-484, University of California, San Diego, June 1996.

[FJL+88] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors*. Prentice-Hall, 1988.

[FWM94] G. C. Fox, R. D. Williams, and P. C. Messina. *Parallel Computing Works*. Morgan Kaufmann, 1994.

[GR96] J. Gehring and A. Reinefeld. Mars: A framework for minimizing the job execution time in a metacomputing environment. *Future Generation Computer Systems*, 12(1):87–99, 1996.

[KDB02] S. Kumar, S. K. Das, and R. Biswas. Graph partitioning for parallel applications in heterogeneous Grid environments. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, 2002.

[LM86] Richard J. Larsen and Morris L. Marx. *An Introduction to Mathematical Statistics and Its Applications*. Prentice-Hall, 1986.

[LYFA02] C. Liu, L. Yang, I. Foster, and D. Angulo. Design and evaluation of a resource selection framework for Grid applications. In *Proceedings of the Eleventh IEEE International Symposium on High-Performance Distributed Computing (HPDC-11)*, 2002.

[MLH95] Shikharesh Majumdar, Johannes Lüthi, and Günter Haring. Histogram-based performance analysis for computer systems with variabilities or uncertainties in workload. Technical Report SCE-95-22, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, November 1995.

[RIF01] M. Ripeanu, A. Iamnitchi, and I. Foster. Performance predictions for a numerical relativity package in Grid environments. *International Journal of High Performance Computing Applications*, 15(4):375–387, 2001.

[SB98] J. Schopf and F. Berman. Performance prediction in production environments. In *Proceedings of Fourteenth International Parallel Processing Symposium and the Ninth Symposium on Parallel and Distributed Processing*, 1998.

[SB99] J. Schopf and F. Berman. Stochastic scheduling. In *Proceedings of SuperComputing (SC'99)*, 1999.

[Sch99] Jennifer M. Schopf. A practical methodology for defining histograms in predictions. In *Proceedings of ParCo '99*, 1999.

[SFT98] W. Smith, I. Foster, and V. Taylor. Predicting application run times using historical information. In D. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing (Proceedings of the Fourth International JSSPP Workshop; LNCS #1459)*. Springer-Verlag, 1998.

[TSC00] H. Turgeon, Q. Snell, and M. Clement. Application placement using performance surface. In *Proceedings of the Ninth IEEE International Symposium on High-Performance Distributed Computing (HPDC-9)*, 2000.

[VS02] S. Vazhkudai and J. M. Schopf. Predicting sporadic Grid data transfers. In *Proceedings of the Eleventh IEEE Symposium on High-Performance Distributed Computing (HPDC-11)*, 2002.

[Wol98] R. Wolski. Dynamically forecasting network performance using the Network Weather Service. *Journal of Cluster Computing*, 1:119–132, January 1998.

[WSH99a] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999.

[WSH99b] R. Wolski, N. Spring, and J. Hayes. Predicting the CPU availability of time-shared Unix systems. In *Proceedings of the Eighth IEEE International Symposium on High-Performance Distributed Computing (HPDC-8)*, 1999.

[WZ98] J.B. Weissman and X. Zhao. Scheduling parallel applications in distributed networks. *Journal of Cluster Computing*, 1:109–118, 1998.

[Yan03] Lingyun Yang. Load traces. `http://cs.uchicago.edu/~lyang/Load`, 2003.

[YFS03] L. Yang, I. Foster, and J. M. Schopf. Homeostatic and tendency-based CPU load predictions. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.