Conservative Scheduling: Using Predicted Variance to Improve Scheduling Decisions in Dynamic Environments

Lingyun Yang[1] Jennifer M. Schopf [2] Ian Foster[1,2]

[1]*Department of Computer Science, University of Chicago, Chicago, IL 60637*
[2]*Math &Computer Science Division, Argonne National Laboratory, Argonne, IL 60439*
lyang@cs.uchicago.edu [jms, foster]@mcs.anl.gov

Abstract

In heterogeneous and dynamic environments, efficient execution of parallel computations can require mappings of tasks to processors with performance that is both irregular (due to heterogeneity) and time-varying (due to dynamicity). While adaptive domain decomposition techniques have been used to address heterogeneous resource capabilities, temporal variations in those capabilities have seldom been considered. 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 techniques, based on time series predictors that we developed in previous work, for predicting CPU load at some future time *point*, average CPU load for some future time *interval,* and *variation* of CPU load over some future time interval. We then present a family of stochastic scheduling algorithms that exploit such predictions of future availability and variability when making data mapping decisions. Finally, 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 due to the presence of other applications sharing the resources. The effective use of this platform, however, requires new approaches to resource scheduling and application mapping capable of dealing with the *heterogeneous* and *dynamic* nature of such systems.

We present here a *conservative scheduling* technique that uses predicted mean and variance CPU capacity information 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 – a cluster may be homogenous in machine type, but quite heterogeneous in performance due to different underlying load on the various resources. Also, 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 a machine with less variance. 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 many other approaches. This technique addresses both the dynamic and heterogeneous nature of shared resources.

We proceed in two steps. First, we extend our previous time series predictor work [31] to obtain three types of prediction: CPU load at some future time *point*, average CPU load for some future time *interval,* and *variation* of CPU load over some future time interval. Then, we extend our previously defined stochastic scheduling algorithms [23] to use the predicted means and variances. 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 [14,15]. We present experiments conducted using Cactus [2-4], a loosely synchronous

iterative [1]computational astrophysics application. These 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.

The rest of this paper is organized as follows. Section 2 introduces related work. Section 3 describes the problem. Section 4 describes our three predictors and Section 5 introduces our conservative scheduling method. Section 6 presents our experimental results. In Section 7, we summarize and briefly discuss future work.

2 Related Work

Many researchers [6,7,12,17-19,27] 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 [7] and Liu et al. [19] use the 10-second-ahead predicted CPU information provided by the Network Weather Service (NWS) [29,30] 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. implemented a user level Real Time Scheduler Advisor (RTSA) [9] , which uses multi-step-ahead CPU load prediction provided by a Running Time Advisor (RTA) (RTA)[8] to instruct workload allocation. While their scheduler uses variation information of the resource availability, the focus is on assisting client applications to meet deadlines for applications running on a single host, not load-balancing between resources.

Yang and Casanova [32,33] developed a multi-round scheduling algorithm for divisible workloads. They use system performance information collected at scheduling time to decide a workload allocation scheme. If the system status changes dramatically during execution of the application, the scheduler is switched to a greedy algorithm that simply assigns more work to idle computers. However, this strategy is limited to application whose subtask are independent each other. While for the loosely synchronous application we concerned in this work, there are communications among subtasks.

Dome [5] and Mars [16] 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.

Schopf and Berman [23] defined a *stochastic scheduling policy* based on time balancing for data-parallel applications. The basic idea of this work is to allocate less work to machines with higher load variance. Their work uses the mean and variation of the history information to instruct the scheduling process. But their algorithm assumes that the associated stochastic data can be described by a normal distribution (because of constraints on the formulas used), which they admit is not always a valid assumption [10,21].

In our approach, we define a time-balancing scheduling strategy based on the 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, and achieve 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 article, we assume that the target set of resources is fixed and 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, or have identical underlying CPU loads. Wihin this context, our goal is to achieve data assignments that balance load

[1] SC'03, November 15-21, 2003, Phoenix, Arizona, USA

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:

$$E_i(D_i) = E_j(D_j) \quad \forall i, j \qquad (1)$$

$$\sum D_i = D_{Total}$$

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 using a performance model of the application. For example, a simple application might have the following performance model:

  $$E_i(D_i) = Comm(D_i)*(futureNWCapacity) + Comp(D_i)*(futureCPUCapacity)$$

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 would be to negotiate a service level agreement (SLA) with the resource owner under which they would contract to provide the specified capability [13]. Or, we could use observed historical data to generate a prediction for future behavior [8,22,24,26,28,30,31]. We focus in this article on the latter approach and present techniques for predicting the future capability. 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) [29,30] 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 [7,19] uses this one-step-ahead predicted CPU information as the future CPU capability in the performance model. However, what is really needed for better data distribution and scheduling is an estimate of the *average* CPU load an application will experience during execution, rather than 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. Thus, a slower machine will not only 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, 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. However, because of its large variation, it is slow to communicate to M2 at the end of the second iteration, which in turn causes the delay of the task on M2 at the third computation step (in black), which causes the delay of 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.

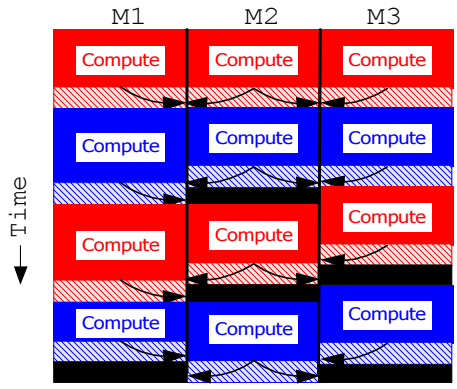


Figure 1: The interrelated influence among tasks of synchronous iterative application.

In our previous work [31], we developed a one step ahead CPU load predictor. We now describe how this 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 4.1); the average interval CPU load for some future time *interval* (Section 4.2); and the *variation* of CPU load over some future time interval (Section 4.3).

4.1 One-Step-Ahead CPU Load Prediction

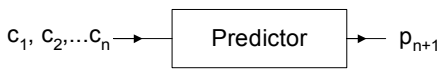
The tendency-based time series predictor developed in our previous work [31] can provide one-step-ahead CPU load prediction based on history CPU load information. This predictor has been demonstrated to be more accurate than other predictors for CPU load data The algorithm predicts the next value according 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 following algorithm, where V_T is the measured value at the T_{th} measurement and P_{T+1} is the predicted value for measurement value V_{T+1}.

```
// 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
        P_{T+1} = V_T + IncrementConstant;
        IncrementConstant adaptation process
else if (Tendency="Decrease") then
        P_{T+1} = V_T – V_T*DecrementFactor;
        DecrementFactor adaptation process
```

We found 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:



where $C=c_1,c_2\ldots c_n$ is the preceding CPU load time series measured at constant-width time interval and p_{n+1} is the predicted value for measurement value c_{n+1}.

4.2 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 [10], 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 non-overlapping 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. If the estimated application execution time is about 100 seconds, the aggregation degree M can be calculated by

M = execution time of application * frequency of original time series (2)
= 100 * 0.1
= 10.

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 is:

$$C= \underbrace{c_1,}\ \ldots\ \underbrace{c_{n-2M+1}\ldots c_{n-M-1},c_{n-M}}\ \underbrace{c_{n-M+1},\ldots c_{n-1},c_n}$$

$$A= a_1\ \ldots\ a_{k-1},\ a_k \qquad k=\lceil n/M \rceil$$

Where

$C=c_1,c_2,\ldots c_n$: the original preceding CPU load time series measured at constant-width time interval;

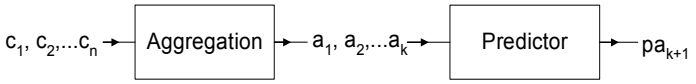
M: the aggregation degree, calculated by Equation 2;

$A=a_1,a_2,\ldots a_k$ ($k=\lceil n/M \rceil$): the interval CPU load time series, calculated by Equation 3:

$$a_i = \sum_{j=1..M} c_{n-(k-i+1)*M+j} \Big/ M \qquad i=1..k \qquad (3)$$

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.

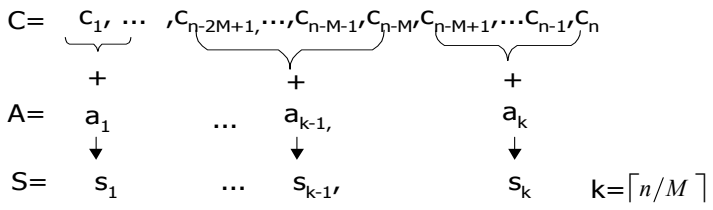
After creating the aggregated time series, the second step of our interval load prediction involves using the one-step-ahead predictor on the aggregated time series to predict the mean interval CPU load.



The output pa_{K+1} is the predicted value of a_{k+1}, which is approximately equal to the average CPU load the application will encounter during execution.

4.3 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 Section 4.2):

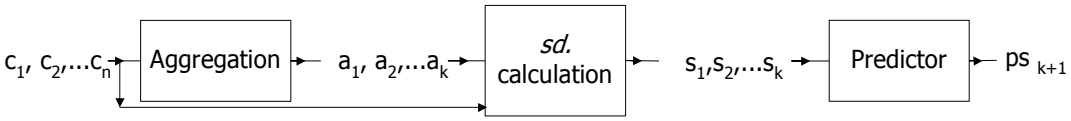


Assuming the original CPU load time series is $C=c_1,c_2,\ldots c_n$, the interval load time series is $A=a_1,a_2,\ldots a_k$($k=\lceil n/M \rceil$), and an aggregation degree of M, we can calculate the standard deviation CPU load time series $S=s_1,s_2,\ldots s_k$:

$$s_i = \sqrt{\sum_{j=1..M} (c_{n-(k-i+1)*M+j} - a_i)^2 / M} \qquad i=1..k \qquad (4)$$

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. We also use the three CPU load predictors introduced above: the next step predicted CPU load at a future time *point* (Section 4.1); the average CPU load for some future time *interval* (Section 4.2); and the *variation* of CPU load over some future time interval (Section 4.3).

5.1 Cactus Application

We applied 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. We use a one-dimensional decomposition to partition the workload in our experiments.

This application is loosely synchronous, as described in Section 4. The full performance model for Cactus is described in [19], but in summary it is:

$$E_i(D_i) = \text{start\_up time} + (D_i * Comp_i(0) + Comm_i(0)) * \text{slowdown(effective CPU load)}.$$

$Comp_i$ (0) and *$comm_i$ (0)*, 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 [20]. 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 [19]:

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:

(1) One Step Scheduling (OSS): Use the one-step-ahead prediction of the CPU load, as described in Section 4.1, for the effective CPU load.

(2) Predicted Mean Interval Scheduling (PMIS): Use the interval load prediction, described in Section 4.2, for the effective CPU load.

(3) Conservative Scheduling (CS): Use the conservative load prediction, equal to the interval load prediction (defined in 4.2) added to a measure of the predicted variance (defined in section 4.3) for the effective CPU load. That is, Effective CPU load= $pa_{k+1} + ps_{k+1.}$

(4) 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 [25,27].

(5) History Conservative Scheduling (HCS): Use the conservative estimate CPU load defined by adding the mean and variance of the history CPU load collected for 5 minutes preceding the application run as the effective CPU load. This approximates the prediction and algorithms used in [23].

6 Conservative Scheduling Experiments

To validate our work, we conducted experiments on workstation clusters at University of Illinois at Champaign-Urbana (UIUC), University of California, San Deigo (UCSD), and Argonne National Laboratory's Chiba City system.

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. At Chiba City, we used a much larger cluster, which includes 32 Linux machines, each with a 500MHz CPU. All machines are dedicated during experiments.

To evaluate the different scheduling polices under identical workloads, we used a load trace playback tool [11] to generate a background workload from a trace of the CPU load that results in realistic and repeatable CPU contention behavior. We chose thirty-two load time series available from [1]. These are all traces of actual

machines, which we characterize by their mean and standard deviation. We used 240 minutes of each trace, at a granularity of 0.1 Hz. The statistic properties of these CPU load traces are shown in Table 1. Note that even though some machines have the same speed, the performance that they deliver to the application varies due to the fact that they each experienced different background loads.

Table 1: The mean and standard deviation of 32 CPU load traces.

CPU Load Trace Name	Machine Name	Mean	SD
LL1	abyss	0.1169 (L)	0.1599 (L)
LL2	axp6	0.0639 (L)	0.1068 (L)
LL3	axp6	0.1803 (L)	0.0944 (L)
LL4	axp6	0.0126 (L)	0.0541 (L)
LL5	axp7	0.0167 (L)	0.0583 (L)
LL6	axp7	0.1631 (L)	0.1042 (L)
LL7	axp7	0.1278 (L)	0.1174 (L)
LL8	axp7	0.0615 (L)	0.1003 (L)
LH1	vatos	0.2199 (L)	0.3101 (H)
LH2	axp1	0.1361 (L)	0.2926 (H)
LH3	axp1	0.2489 (L)	0.3436 (H)
LH4	axp1	0.1909 (L)	0.2912 (H)
LH5	axp1	0.3917 (L)	0.3839 (H)
LH6	axp2	0.2298 (L)	0.2830 (H)
LH7	axp2	0.1778 (L)	0.2831 (H)
LH8	axp2	0.1601 (L)	0.2996 (H)
HL1	Mystere	1.8489 (H)	0.1432 (L)
HL2	pitcairn	1.1884 (H)	0.1221 (L)
HL3	axp4	1.1112 (H)	0.1289 (L)
HL4	axp4	1.1797 (H)	0.1241 (L)
HL5	axp4	1.1035 (H)	0.1199 (L)
HL6	axp4	1.1738 (H)	0.1113 (L)
HL7	axp4	1.0451 (H)	0.0968 (L)
HL8	axp4	1.1061 (H)	0.1250 (L)
HH1	axp0	1.0725 (H)	0.4793 (H)
HH2	axp0	1.0503 (H)	0.4552 (H)
HH3	axp0	1.0315 (H)	0.4958 (H)
HH4	axp0	0.9856 (H)	0.5718 (H)
HH5	axp10	1.1811 (H)	0.3105 (H)
HH6	axp10	1.2839 (H)	0.3224 (H)
HH7	axp10	1.2350 (H)	0.3258 (H)
HH8	axp10	1.2456 (H)	0.3528 (H)

6.2 Experimental Results

Results from six representative experiments are shown in Figures 2–7. A summary of the testbeds and the CPU load traces used for the experiments is given in Table 2.

Table 2: CPU load traces used for every experiment

Experiments	Testbed	CPU Load Traces	Execution Time
Figure 2	UIUC	LL1, LH1, HL1, HH1	Short (≈ 1 minute)
Figure 3	UIUC	LL1, LL2, LH1, LH2	Short (≈ 1 minute)
Figure 4	UIUC	HL2, HL3, HH1, HH2	Short (≈ 1 minute)
Figure 5	UIUC	LH1, LH2, HH1, HH2	Short (≈ 1 minute)
Figure 6	UCSD	LL1, LL2, LH1, LH2, HL1, HL2	Short (≈ 1 minute)
Figure 7	UCSD	LH1, LH2, HL1, HL2, HH1, HH2	Short (≈ 1 minute)
Figure 8	UCSD	LH1, LH2, HL1, HL2, HH1. HH2	Long (≈ 10 minutes)
Figure 9	Chiba City	All 32 load traces	Long (≈ 10 minutes)

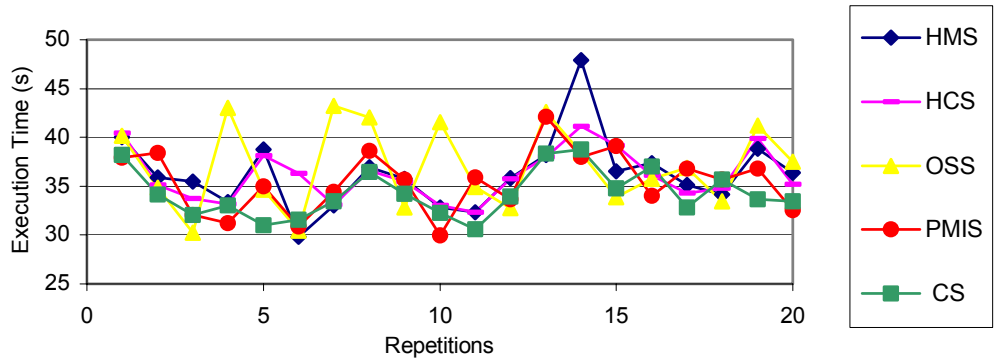


Figure 2: Comparison of the History Mean, History Conservative, One-Step, Predicted Mean Interval and Conservative Scheduling policies, on the UIUC cluster with two low-variance machines (one with low mean, the other with relatively high mean) and two high-variance machines (one with low mean, the other with high mean). The application execution time is about 1 minute.

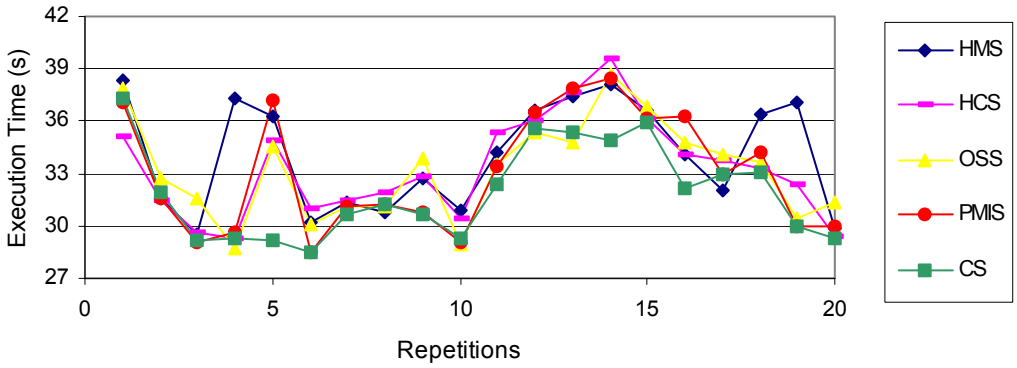


Figure 3: Comparison of the History Mean, History Conservative, One-Step, Predicted Mean Interval and Conservative Scheduling policies, on the UIUC cluster with two low-variance and two high-variance machines (all have low mean). The application execution time is about 1 minute.

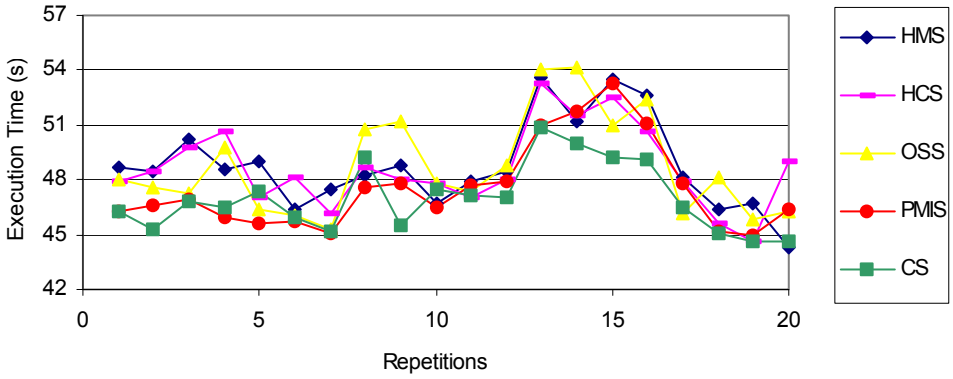


Figure 4: Comparison of the History Mean, History Conservative, One-Step, Predicted Mean Interval and Conservative Scheduling policies, on the UIUC cluster with two low-variance and two high-variance machines (all have high mean). The application execution time is about 1 minute.

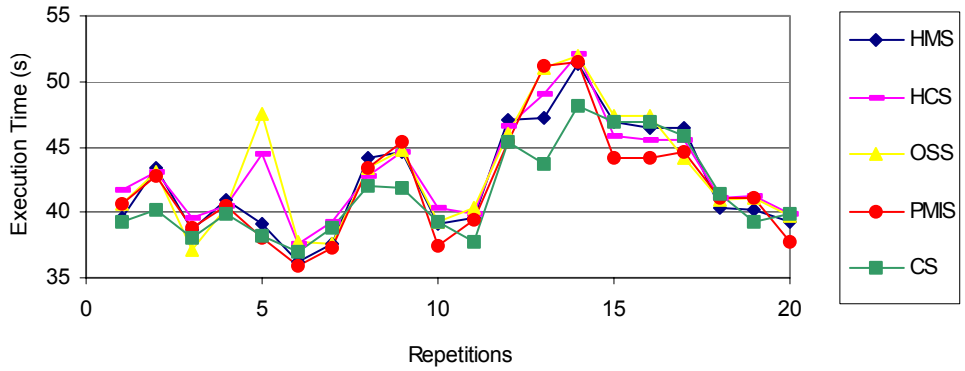


Figure 5: Comparison of the History Mean, History Conservative, One-Step, Predicted Mean Interval, Conservative Scheduling policies, on the UIUC cluster with four high variance machines (two with high, two with low mean). The application execution time is about 1 minute.

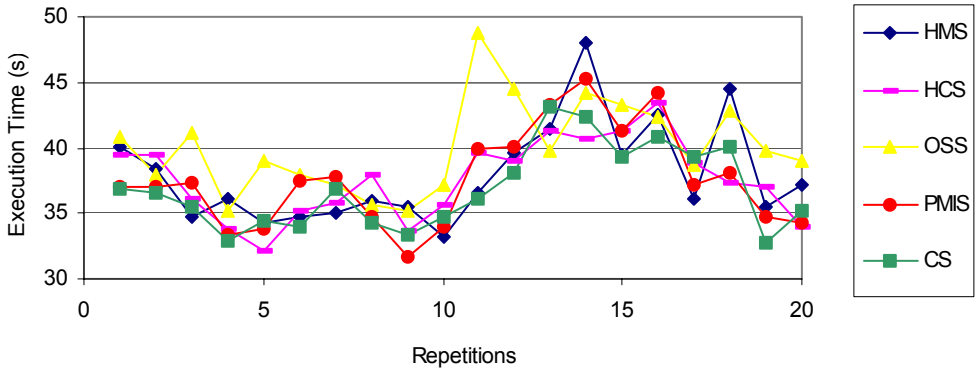


Figure 6: Comparison of the History Mean, History Conservative, One-step, Predicted Mean Interval, Conservative Scheduling policies, on the heterogeneous UCSD cluster with four low-variance machines (two with low mean, two with high mean) and two high-variances machines (have low mean). The application execution time is about 1 minute.

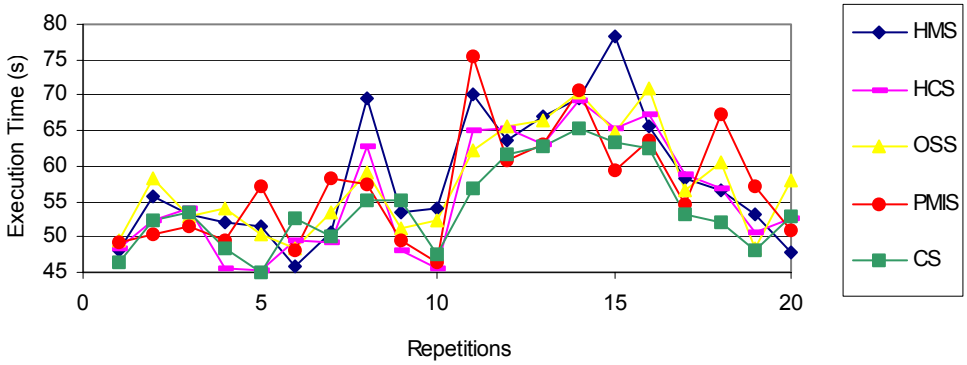


Figure 7: Comparison of the History Mean, History Conservative, One-step, Predicted Mean Interval, Conservative Scheduling policies, on the heterogeneous UCSD cluster with two low-variance machines (have high mean) and four high-variances machines (two with low mean, two with high mean). The application execution time is about 1 minute.

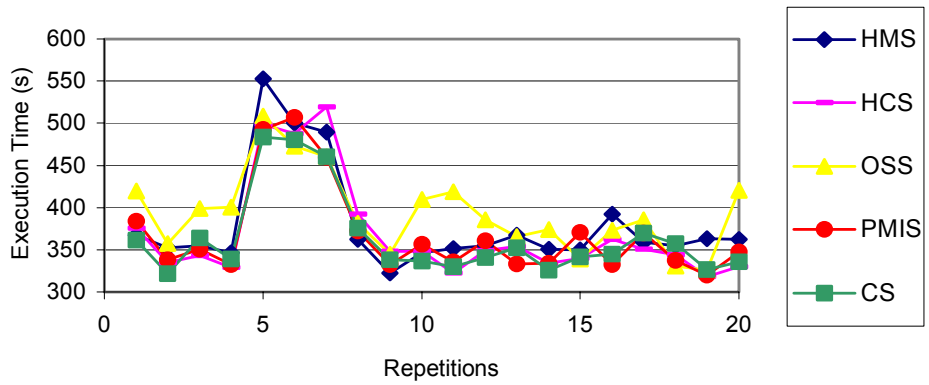


Figure 8: Comparison of the History Mean, History Conservative, One-step, Predicted Mean Interval, Conservative Scheduling policies, on the heterogeneous UCSD cluster within two low-variance machines (have high mean) and four high-variances machines (two with low mean, two with high mean). The application execution time is about 10 minutes.

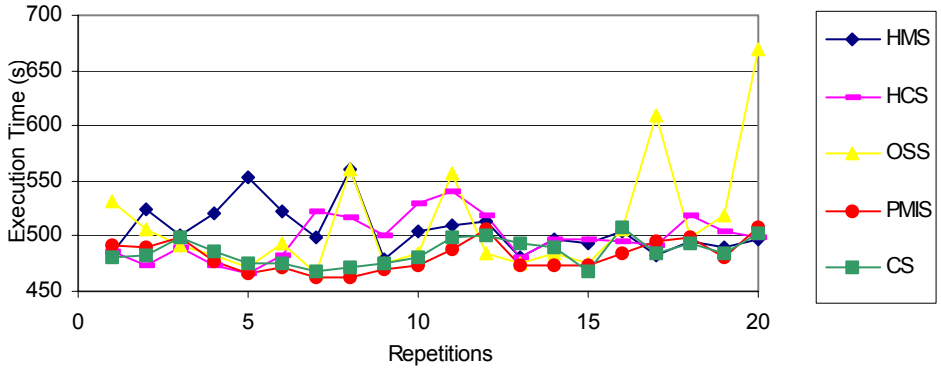


Figure 9 Comparison of the History Mean, History Conservative, One-step, Predicted Mean Interval, Conservative Scheduling policies, on the Chiba City cluster within sixteen low-variance machines (eight with high mean, eight with low mean) and sixteen high-variances machines (eight with low mean, eight with high mean). The application execution time is about 10 minutes.

To compare these policies, we used three metrics: an absolute comparison of run times, a relative measure of achievement and the statistical analysis to show the improvement of our strategy. 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 3. This metric gives a rough valuation on the performance of each scheduling policy over a given interval of time. We can see from Table 3 that 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%–8% less overall execution time than 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 2%–29% less standard deviation of execution time than the History Mean. The Conservative Scheduling policy exhibited 1.5%-77% less standard deviation in execution time than the One-Step Scheduling policy and 7%–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).

Table 3: Average mean and average standard deviation for entire set of runs for each scheduling policy, with the best in each experiment shown in boldface.

Experiment	HMS		HCS		OSS		PMIS		CS	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD
Figure 2	36.23	3.72	36.09	2.62	37.02	4.16	35.44	3.15	**34.26**	**2.44**
Figure 3	34.07	3.10	33.29	2.83	33.20	2.73	33.06	3.37	**31.94**	**2.69**
Figure 4	47.76	2.44	48.63	2.22	48.72	2.70	47.56	2.40	**46.99**	**1.83**
Figure 5	42.28	4.02	42.87	3.62	42.87	4.24	41.96	4.28	**41.26**	**3.24**
Figure 6	37.95	3.83	37.58	**3.02**	37.76	3.52	37.63	3.78	**36.84**	3.12
Figure 7	58.23	9.08	55.74	8.12	57.67	7.17	57.04	8.02	**54.23**	**6.09**
Figure 8	380.00	60.10	369.11	59.05	393.70	**47.78**	368.29	54.10	**363.33**	50.06
Figure 9	505.66	22.07	499.28	20.22	512.10	51.92	**484.22**	14.13	485.79	**11.86**

These results are given in Table 4, 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.

The third metric involves using T-test value to show the significance of the improvement of our strategy over other strategies. We calculated both paired and unpaired one-tailed T-test values for our strategy and each other strategies. The results are shown in Table 5 and Table 6 respectively. We can see that most of the T-test values, especially the paired T-test values, are blow 10%. These results indicate that the possibility of the improvement happening by chance is quite small. So we can say that our Conservative Scheduling policy achieves significant improvements over other four strategies in most cases.

To summarize our results: independent of the loads, CPU capabilities, the execution time of application and number of resources 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 Conclusion 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 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 by extending our earlier work on time series predictors. 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 technique can obtain better execution times and more predictable application behavior than previous methods that focused on predicted means alone, or that used variances in less effective manner. 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.

Table 4: Summary statistics using *Compare* to evaluate five scheduling policies, with the largest value in each case shown in boldface

Experiment	Policy	Best	Good	Avg	Poor	Worst
Figure 2	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
Figure 3	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
Figure 4	HMS	1	3	1	**6**	**9**
	HCS	2	1	**9**	3	5
	OSS	1	2	6	5	6
	PMIS	7	6	3	4	0
	CS	**9**	**8**	1	2	0
Figure 5	HMS	1	**8**	4	1	**6**
	HCS	0	3	**6**	5	**6**
	OSS	2	3	4	**7**	4
	PMIS	7	3	4	4	2
	CS	**10**	3	2	3	2
Figure 6	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
Figure 7	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
Figure 8	HMS	2	3	2	**9**	4
	HCS	**6**	3	**8**	0	3
	OSS	3	0	2	5	10
	PMIS	3	**8**	4	3	2
	CS	**6**	6	4	3	1
Figure 9	HMS	2	1	5	**5**	**7**
	HCS	4	4	2	4	6
	OSS	1	4	**6**	4	5
	PMIS	**8**	4	3	**5**	0
	CS	5	**7**	4	2	2

Table 5: Paired one-tailed T test value for Conservative Scheduling policy and other four policies. (* means no improvement)

Experiments	HMS	HCS	OSS	PMIS
Figure2	0.25%	0.09%	0.37%	2.60%
Figure 3	0.07%	0.17%	0.11%	1.27%
Figure 4	<0.01%	<0.01%	<0.01%	4.30%
Figure 5	0.71%	0.16%	1.01%	17.76%
Figure 6	1.83%	6.76%	0.02%	3.43%
Figure 7	0.24%	4.40%	0.01%	3.75%
Figure 8	0.19%	12.05%	0.03%	12.72%
Figure 9	0.21%	0.69%	1.41%	*

Table 6: Unpaired one-tailed T test value for conservative scheduling policy and other four policies. (* means no improvement)

Experiments	HMS	HCS	OSS	PMIS
Figure2	2.79%	1.42%	0.83%	9.83%
Figure 3	1.32%	6.60%	7.77%	12.78%
Figure 4	0.65%	0.73%	1.14%	20.25%
Figure 5	23.32%	9.51%	10.90%	33.60%
Figure 6	15.94%	22.57%	0.22%	23.69%
Figure 7	5.51%	25.48%	5.51%	10.99%
Figure 8	18.80%	39.32%	3.27%	40.58%
Figure 9	0.05%	0.71%	1.65%	*

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, U.S. Department of Energy, under contract W-31-109-Eng-38.

References

[1] http://cs.uchicago.edu/~lyang/Load.

[2] Allen, G., Benger, W., Goodale, T., Hege, H.-C., Lanfermann, G., Merzky, A., Radke, T., Seidel, E. and Shalf, J., Cactus Tools for Grid Applications, *Cluster Computing*, 4 (2001) 179-188.

[3] Allen, G., Benger, W., Goodale, T., Hege, H.-C., Lanferrmann, G., Merzky, A., Radke, T., Seidel, E. and Shalf, J., The Cactus Code: A Problem Solving Environment for the Grid. *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC9)*, Pittsburgh, 2000.

[4] Allen, G., Goodale, T., Lanfermann, G., Radke, T., Seidel, E., Benger, W., Hege, H.-C., Merzky, A., Masso, J. and Shalf, J., Solving Einstein's Equations on Supercomputers, *IEEE Computer*, 32 (1999) 52-59.

[5] Arabe, J.N.C., Beguelin, A., Loweamp, B., Seligman, E., Starkey, M. and Stephan, P., Dome: Parallel Programming in a Heterogeneous Multi-user Environment. Carnegie Mellon University, School of Computer Science, 1995.

[6] Berman, F., Wolski, R., Figueira, S., Schopf, J. and Shao, G., Application-Level Scheduling on Distributed Heterogeneous Networks. *Supercomputing'96*, 1996.

[7] Dail, H.J., A Modular Framework for Adaptive Scheduling in Grid Application Development Environments. *Computer Science*, University of California, California, San Diego, 2001.

[8] Dinda, P.A., Online Prediction of the Running Time of Tasks. *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC 2001)*, San Francisco, CA, 2001.

[9] Dinda, P.A., A Prediction-based Real-time Scheduling Advisor. *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS 2002)*, 2002.

[10] Dinda, P.A. and O'Hallaron, D.R., The Statistical Properties of Host Load. *The Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR 98)*, Pittsburgh, PA, 1998, pp. 319-334.

[11] Dinda, P.A. and O'Hallaron, D.R., Realistic CPU Workloads Through Host Load Trace Playback. *Proc. 5th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR 2000)*, *Vol. Springer LNCS 1915*, Rochester, NY, 2000, pp. 265-280.

[12] Figueira, S.M. and Berman, F., Mapping Parallel Applications to Distributed Heterogeneous Systems. University of Californian, San Diego, 1996.

[13] Foster, I. and Kesselman, C., GriPhyN/PPDG: Data Grid Architecture, Toolkit, and Roadmap. The GriphyN and PPDG Collaborations, 2001, pp. 31.

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

[15] Fox, G.C., Williams, R.D. and Messina, P.C., *Parallel Computing Works*, Morgan Kaufmann, 1994, 977 pp.

[16] Gehring, J. and Reinefeld, A., Mars: A Framework for Minimizing the Job Execution Time in a Metacomputing Environment, *Future Generation Computer Systems*, 12(1) (1996) 87-99.

[17] Kumar, S., Das, S.K. and Biswas, R., Graph Partitioning for Parallel Applications in Heterogeneous Grid Environments. *submitted to International Parallel and Distributed Processing Symposium (IPDPS 2002)*, Florida, 2002.

[18] Kumar, S., Maulik, U., Bandyopadhyay, S. and Das, S.K., Efficient Task Mapping on Distributed Heterogeneous System for Mesh Applications. *International workshop on Distributed Computing (IWDC 2001)*, Calcutta, India, 2001.

[19] Liu, C., Yang, L., Foster, I. and Angulo, D., Design and Evaluation of a Resource Selection Framework for Grid Applications. *Proceedings of the 11th IEEE International Symposium on High-Performance Distributed Computing (HPDC 11)*, Edinburgh, Scotland, 2002.

[20] Ripeanu, M., Iamnitchi, A. and Foster, I., Performance Predictions for a Numerical Relativity Package in Grid Environments, *International Journal of High Performance Computing Applications*, 15 (2001).

[21] Schopf, J.M., Performance Prediction and Scheduling for Parallel Applications on Multi-User Cluster. *Department of Computer Science and Engineering*, University of California San Diego, San Diego, 1998, pp. 247.

[22] Schopf, J.M., A Practical Methodology for Defining Histograms for Predictions and Scheduling. *ParCo'99*, 1999.

[23] Schopf, J.M. and Berman, F., Stochastic Scheduling. *SuperComputing'99*, Portland, Oregon, 1999.

[24] Smith, W., Foster, I. and Taylor, V., Predicting Application Run Times Using Historical Information. *Proceedings of the IPPS/SPDP'98 workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[25] Turgeon, A., Snell, Q. and Clement, M., Application Placement Using Performance Surface. *HPDC2000*, Pittsburgh, Pennsylvania, 2000.

[26] Vazhkudai, S., Schopf, J.M. and Foster, I., Predicting the Performance of Wide Area Data Transfer. *16th Int'l Parallel and Distributed Processing Symposium (IPDPS 2002)*, Fort Lauderdale, Florida, 2002.

[27] Weissman, J.B. and Zhao, X., Scheduling Parallel Applications in Distributed Networks, *Journal of Cluster Computing*, 1 (1998) 109-118.

[28] Wolski, R., Dynamically Forecasting Network Performance Using the Network Weather Service, *Journal of Cluster Computing* (1998).

[29] Wolski, R., Spring, N. and Hayes, J., The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing, *Journal of Future Generation Computing Systems* (1998) 757-768.

[30] Wolski, R., Spring, N. and Hayes, J., Predicting the CPU availability of Time-shared Unix Systems. *Proceedings of 8th IEEE High Performance Distributed Computing Conference (HPDC8)*, Redondo Beach, CA, 1999.

[31] Yang, L., Foster, I. and Schopf, J.M., Homeostatic and Tendency-based CPU Load Predictions. *International Parallel and Distributed Processing Symposium (IPDPS2003)*, Nice, France, 2003.

[32] Yang, Y. and Casanova, H., RUMR: Robust Scheduling for Divisible Workloads. *Proceedings of the 12th IEEE Symposium on High Performance and Distributed Computing (HPDC-12)*, Seattle, 2003.

[33] Yang, Y. and Casanova, H., UMR: A Multi-Round Algorithm for Scheduling Divisible Workloads. *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)*, Nice, France, 2003.