

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

Lingyun Yang¹ Jennifer M. Schopf² Ian Foster^{1,2}

¹*Department of Computer Science, University of Chicago, Chicago, IL 60637*

²*Mathematics and 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 whose performance is both irregular (because of heterogeneity) and time-varying (because of 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 both 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. The effective use of such platforms thus 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. We emphasize that a cluster may be homogenous in machine type but quite heterogeneous in performance because of 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 do. This technique addresses both the dynamic and the heterogeneous nature of shared resources.

We proceed in two steps. First, we extend our previous time series predictor work [32] 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 [25] to use the predicted means and variances. The result is an approach that exploits

* SC'03, November 15-21, 2003, Phoenix, Arizona, USA
Copyright 2003 ACM 1-58113-695-1/03/0011...\$5.00

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 [16,17]. We present experiments conducted using Cactus [3-5], a loosely synchronous iterative 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 our work.

2 Related Work

Many researchers [7,9,15,19-21,29] 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 [9] and Liu et al. [21] use the 10-second-ahead predicted CPU information provided by the Network Weather Service (NWS) [30,31] 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. use multiple-step-ahead predictions of host load [13] and their associated error covariance information to predict the running times of tasks as confidence intervals [10]. These confidence intervals, a representation of prediction error variances, can then be used for various scheduling goals, for example, soft real-time [11]. In contrast, we predict the variance of host load itself and focus on scheduling iterative data parallel tasks to minimize overall run-time.

Yang and Casanova [33,34] developed a multiround 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 subtasks are independent each other. For the loosely synchronous application we are concerned with in this work, there are communications among subtasks.

Dome [6] and Mars [18] support dynamic workload balancing through migration and make the application adaptive to the dynamic environment at run time. But the implementation of such adaptive strategies can be complex and is not feasible for all applications.

Schopf and Berman [25] 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 [12,23].

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 to 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 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 or 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:

$$\begin{aligned} E_i(D_i) &= E_j(D_j) \quad \forall i, j \\ \sum D_i &= D_{Total} \end{aligned} \quad (1)$$

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:

$$E_i(D_i) = \text{Comm}(D_i) * (\text{futureNWCapacity}) + \text{Comp}(D_i) * (\text{futureCPUCapacity}).$$

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

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 owners under which they would contract to provide the specified capability [8]. Or, we could use observed historical data to generate a prediction for future behavior [10,24,26,28,30-32]. 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) [30,31] 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 [9,21] 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. Because of its large variation, however, 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.

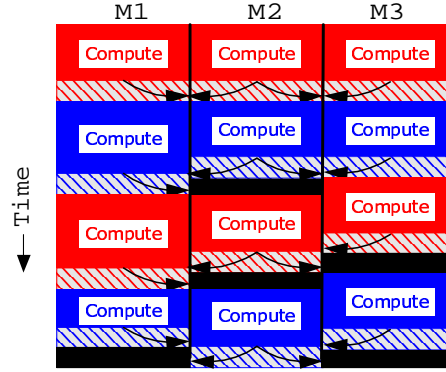


Figure 1: The interrelated influence among tasks of synchronous iterative application.

In our previous work [32], 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

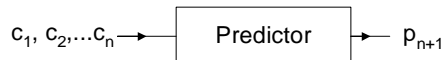
The tendency-based time series predictor developed in our previous work [32] can provide one-step-ahead CPU load prediction based on history CPU load information. This predictor has been demonstrated to be more accurate than other predictors in some situations. The algorithm predicts the next value according 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
```

Test indicates that a mixed-variation (that is, different behavior for the increment from that of the decrement) experimentally performs 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 $C=c_1, c_2, \dots, c_n$ 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 [12], 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. If the estimated application execution time is about 100 seconds, the aggregation degree M can be calculated by

$$\begin{aligned} M &= \text{execution time of application} * \text{frequency of original time series} & (2) \\ &= 100 * 0.1 \\ &= 10. \end{aligned}$$

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

$$\begin{array}{l} C = \underbrace{c_1, \dots, c_{n-2M+1}}_{a_{k-1}}, \dots, \underbrace{c_{n-M+1}, \dots, c_{n-1}}_{a_k}, c_n \\ A = a_1, \dots, a_{k-1}, \dots, a_k \quad k = \lceil n/M \rceil \end{array}$$

where

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

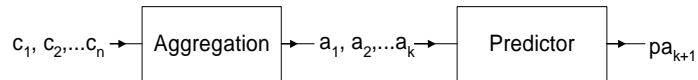
M : the aggregation degree, calculated by Equation 2;

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

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

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):

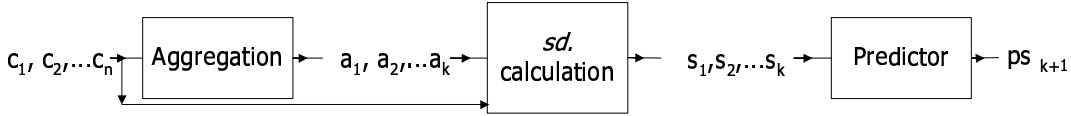
$$\begin{array}{r}
 C = \underbrace{c_1, \dots, c_{n-2M+1}, \dots, c_{n-M-1}, c_{n-M}}_{+} \underbrace{c_{n-M+1}, \dots, c_{n-1}, c_n}_{+} \\
 A = a_1 \quad \dots \quad a_{k-1}, \quad \dots \quad a_k \\
 S = s_1 \quad \dots \quad s_{k-1}, \quad \dots \quad s_k \quad k = \lceil n/M \rceil
 \end{array}$$

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

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

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 elsewhere [21], but in summary it is

$$E_i(D_i) = \text{start_up time} + (D_i * \text{Comp}_i(0) + \text{Comm}_i(0)) * \text{slowdown}(\text{effective CPU load}).$$

$Comp_i(0)$ 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 [22]. We incur a startup time when initiating computation on multiple processors in a workstation cluster that is 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 [21].

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 job execution time, 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 Section 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 [27,29].
- (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 [25].

6 Conservative Scheduling Experiments

To validate our work, we conducted experiments on workstation clusters at the University of Illinois at Champaign-Urbana (UIUC), University of California, San Diego (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 450 MHz Linux machines. 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 of thirty-two 500MHz CPU Linux machines. All machines were dedicated during experiments.

To evaluate the different scheduling policies under identical workloads, we used a load trace playback tool [14] to generate a background workload from a trace of the CPU load that results in realistic and repeatable CPU contention behavior. We chose sixty-four load time series available from [1]. We used 240 minutes of each trace, at a granularity of 0.1 Hz. These are all traces of actual machines, which we characterize by their mean and standard deviation. We consider a load trace having a *high* mean if its mean is greater than 0.9; or it has a *low* mean if its mean is less than 0.5. We consider a load trace having a *high* variation if its standard deviation is greater than 0.28; or it has a *low* variation if its standard deviation is less than 0.16. So the sixty-four time series include sixteen low-mean, low-variation load traces (represented by LL); sixteen low-mean, high-variation load traces (represented by LH); sixteen high-mean, low variation load traces (represented by HL); and sixteen high-mean, high-variation load traces (represented by HH). The detailed statistic properties of these CPU load traces can be found in [1]. Note that even though some machines have the same speed, the performance that they deliver to the application varies because they each experienced different background loads.

6.2 Experimental Results

Results from ten representative experiments are shown in Figures 2–11. The testbeds and the CPU load traces used for the experiments are summarized in Table 1.

Table 1: CPU load traces used for every experiment

Experiments	Testbed	CPU Load Traces	Execution Time
Figure 2	UIUC	one LL , one LH , one HL and one HH load traces	Short (\approx 1 minute)
Figure 3	UIUC	two LL and two LH load traces	Short (\approx 1 minute)
Figure 4	UIUC	two HL and two HH load traces	Short (\approx 1 minute)
Figure 5	UIUC	two LH and two HH load traces	Short (\approx 1 minute)
Figure 6	UCSD	two LL, two LH and two HL load traces	Short (\approx 1 minute)
Figure 7	UCSD	two LH, two HL and two HH load traces	Short (\approx 1 minute)
Figure 8	UCSD	two LH, two HL and two HH load traces	Long (\approx 10 minutes)
Figure 9	Chiba City	eight LL, eight LH, eight HL and eight HH load traces	Long (\approx 10 minutes)
Figure 10	Chiba City	sixteen LL and sixteen HL load traces	Long (\approx 10 minutes)
Figure 11	Chiba City	sixteen LH and sixteen HH load traces	Long (\approx 10 minutes)

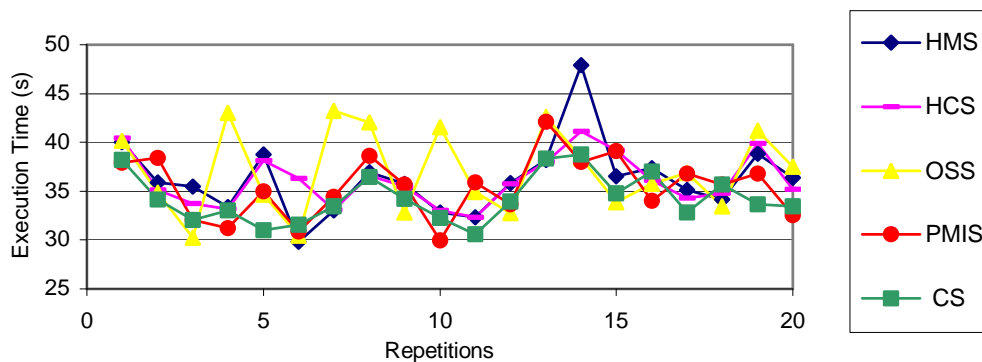


Figure 2: Comparison of the History Mean, History Conservative, One-Step, Predicted Mean Interval, and Conservative Scheduling policies, on the UIUC cluster with two low-variance machines (one with low mean, the other with relatively high mean) and two high-variance machines (one with low mean, the other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

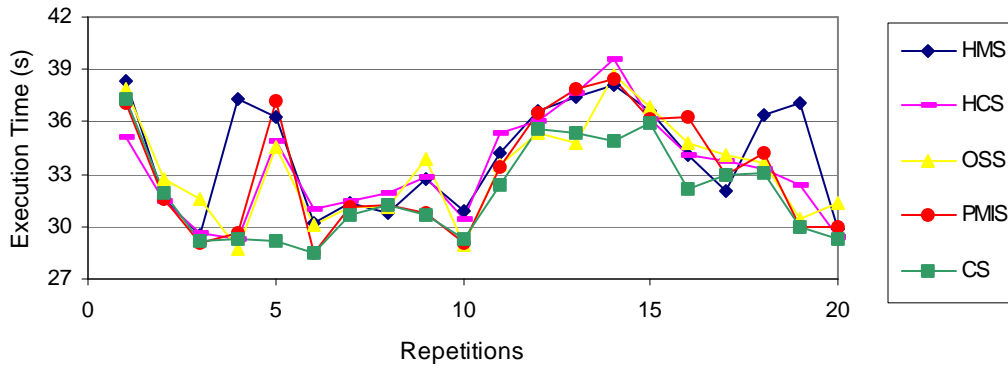


Figure 3: Comparison of the History Mean, History Conservative, One-Step, Predicted Mean Interval, and Conservative Scheduling policies, on the UIUC cluster with two low-variance and two high-variance machines (all have low mean). The application execution time is about 1 minute.

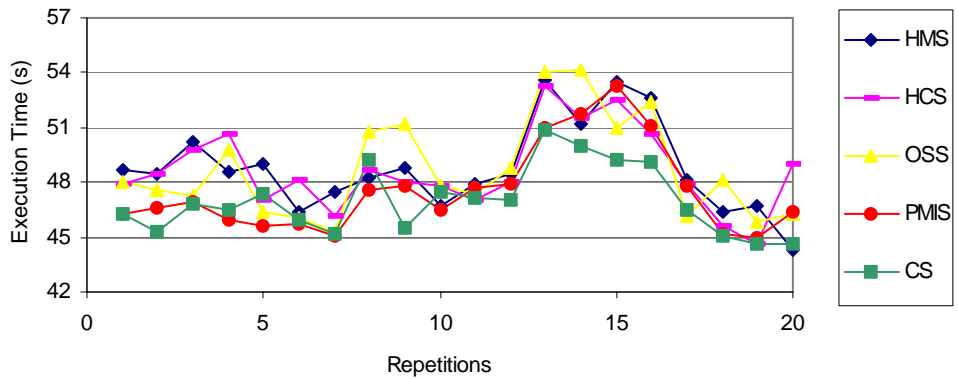


Figure 4: Comparison of the History Mean, History Conservative, One-Step, Predicted Mean Interval, and Conservative Scheduling policies, on the UIUC cluster with two low-variance and two high-variance machines (all have high mean). The application execution time is about 1 minute.

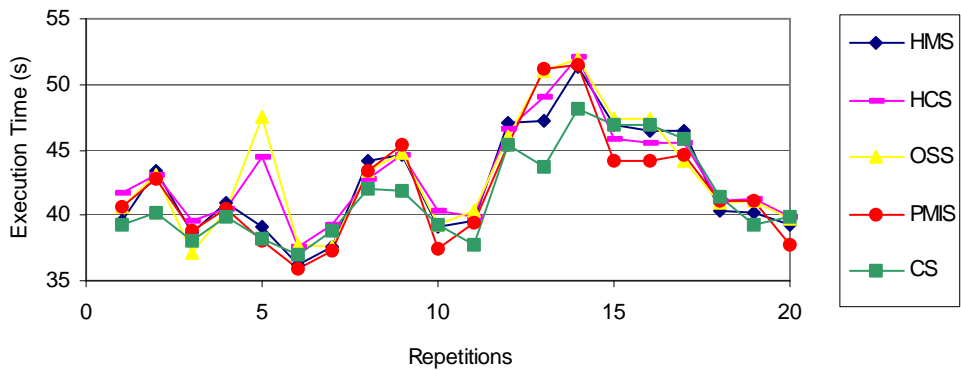


Figure 5: Comparison of the History Mean, History Conservative, One-Step, Predicted Mean Interval, and Conservative Scheduling policies, on the UIUC cluster with four high-variance machines (two with high, two with low mean). The application execution time is about 1 minute.

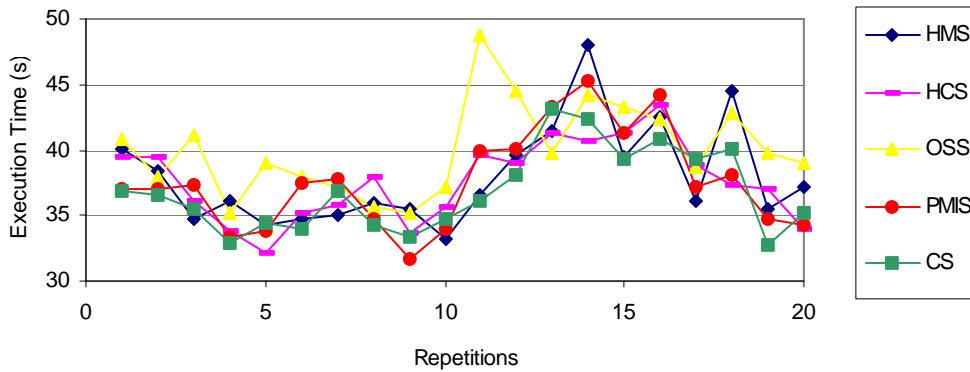


Figure 6: Comparison of the History Mean, History Conservative, One-step, Predicted Mean Interval, and Conservative Scheduling policies, on the heterogeneous UCSD cluster with four low-variance machines (two with low mean, two with high mean) and two high-variance machines (have low mean). The application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

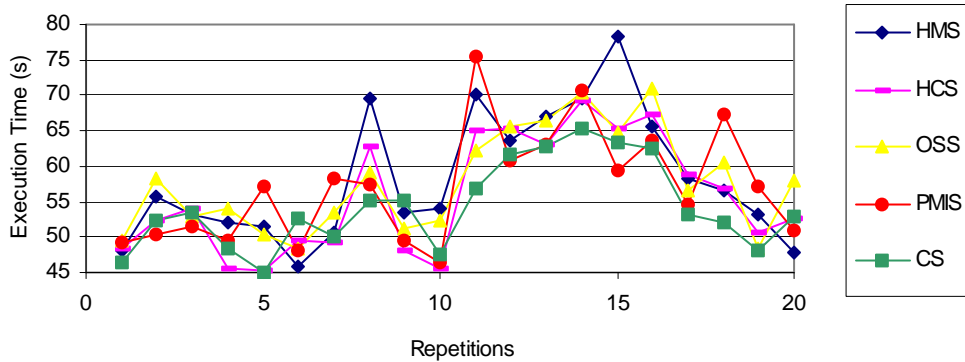


Figure 7: Comparison of the History Mean, History Conservative, One-step, Predicted Mean Interval, and Conservative Scheduling policies, on the heterogeneous UCSD cluster with two low-variance machines (have high mean) and four high-variance machines (two with low mean, two with high mean). The application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

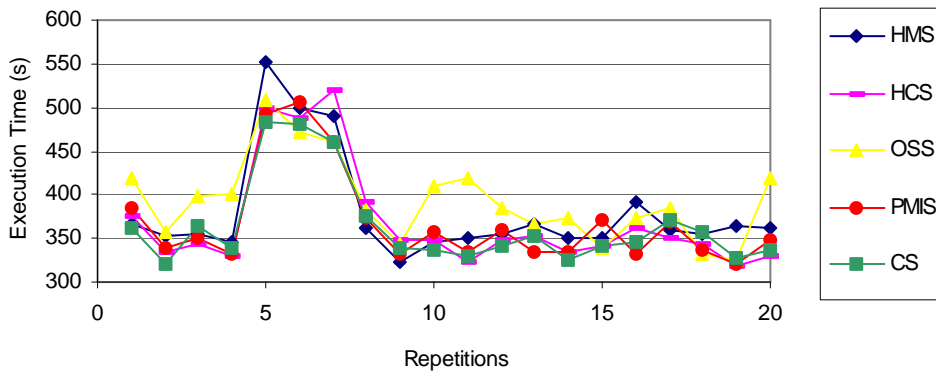


Figure 8: Comparison of the History Mean, History Conservative, One-step, Predicted Mean Interval, and Conservative Scheduling policies, on the heterogeneous UCSD cluster with two low-variance machines (have high mean) and four high-variance machines (two with low mean, two with high mean). The application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

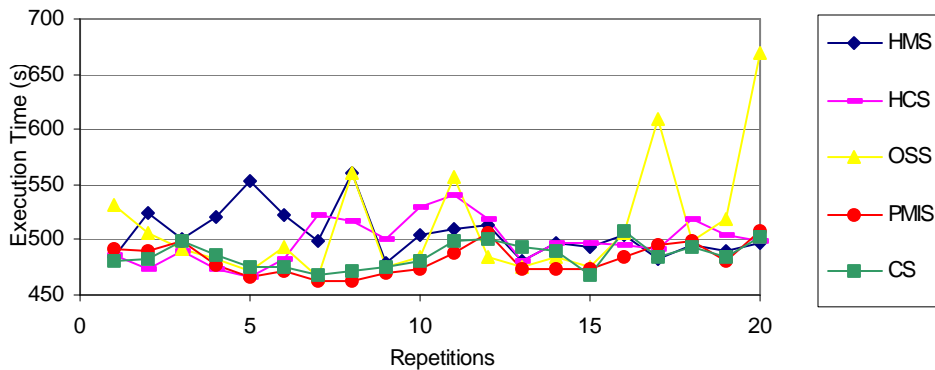


Figure 9: Comparison of the History Mean, History Conservative, One-step, Predicted Mean Interval, and Conservative Scheduling policies, on the Chiba City cluster with sixteen low-variance machines (eight with high mean, eight with low mean) and sixteen high-variance machines (eight with low mean, eight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

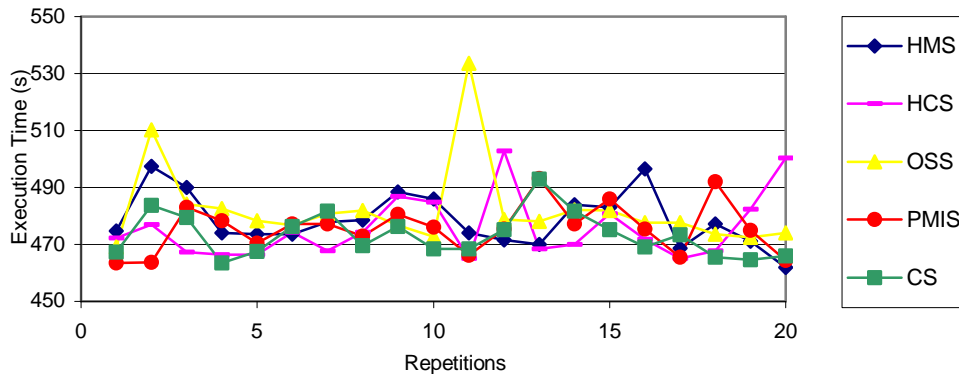


Figure 10: Comparison of the History Mean, History Conservative, One-step, Predicted Mean Interval, and Conservative Scheduling policies, on the Chiba City cluster with thirty-two low-variance machines (sixteen with high mean, sixteen with low mean). The application execution time is about 10 minutes.

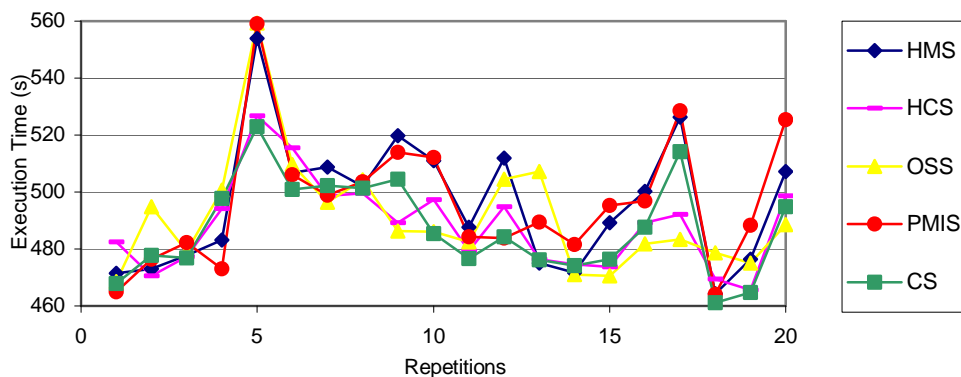


Figure 11: Comparison of the History Mean, History Conservative, One-step, Predicted Mean Interval, and Conservative Scheduling policies, on the Chiba City cluster with thirty-two high-variance machines (sixteen with high mean, sixteen with low mean). The application execution time is about 10 minutes.

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 significance of the improvement of our strategy. The first metric involves an average mean and an average standard deviation for the set of run times of each scheduling policy as a whole, as shown in Table 2. This metric gives a rough valuation on the performance of each scheduling policy over a given interval of time. We can see from Table 2 that over the entire run, the Conservative Scheduling policy exhibited 2%–7% less overall execution time than did the 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%–32% less standard deviation of execution time than did the History Mean. The Conservative Scheduling policy exhibited 1.5%–77% less standard deviation in execution time than did the One-Step Scheduling policy and 7%–41% less standard deviation of execution time than did the Predicted Mean Interval Scheduling policy.

Table 2: 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
Figure 10	478.63	9.38	475.57	11.10	482.15	14.65	475.69	8.67	473.28	7.70
Figure 11	495.89	22.88	488.28	15.38	491.40	20.39	496.41	23.30	487.39	16.81

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 3, 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 the T-test to show the significance of the improvement of our strategy over other strategies. The T-test is a statistical method used to assess whether the means of two groups are significantly different from each other [2]. The result of a T-test is a set of P-values that indicate the possibility that the differences could have happened by chance: a lower P-value means a more significant difference between two groups, so for our experiments smaller numbers are better. T-tests can be paired or unpaired; a paired T-test is used when the two groups are not independent, and an unpaired test is used when the two groups are independent. For our experiments, we calculate both paired and unpaired T-tests because it was not always clear whether the groups should be considered independent of one another. In addition, T-tests can be one-tailed, which is used when one group is expected to always be less than (or greater than) the other and we know that direction, or two-tailed, which is used only to show a difference that can sometimes be less and sometimes be greater. Since our strategy should always be better than the other strategies, we use a one-tail test.

Table 3: 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
Figure 10	HMS	3	2	3	6	6
	HCS	7	6	1	3	3
	OSS	0	2	5	7	6
	PMIS	2	6	6	2	4
	CS	8	4	5	2	1
Figure 11	HMS	1	3	5	6	5
	HCS	2	8	6	2	2
	OSS	7	1	3	3	6
	PMIS	3	2	2	6	7
	CS	7	6	4	3	0

The results of the paired and unpaired one-tailed T-tests comparing the Conservative strategy with the other four strategies are shown in Table 4 and Table 5, respectively, with P-values smaller than 10% shown in boldface. We see that most P-values, especially those for paired T-test, are below 10%. These results indicate that the possibility of the improvement happening by chance is quite small. Thus, we conclude that our conservative scheduling policy achieves significant improvements relative to the other four strategies in most cases.

To summarize our results: independent of the loads, CPU capabilities, application execution time, and number of resources, 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 all clusters, and the policy that resulted in the shortest execution time and the smallest variation in execution time.

Table 4: Paired one-tailed T test value for the conservative scheduling policy relative to each of 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%	*
Figure 10	1.71%	23.90%	0.98%	13.4%
Figure 11	0.16%	33.31%	13.21%	0.44%

Table 5: Unpaired one-tailed T test value for the conservative scheduling policy relative to each of the 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%	*
Figure 10	2.79%	22.56%	1.07%	17.83%
Figure 11	9.43%	43.17%	25.07%	8.44%

7 Conclusion

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 obtains better execution times and more predictable application behavior than previous methods that focused on predicted means alone, or that used variances in a 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.

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

- [1] <http://cs.uchicago.edu/~lyang/Load>.
- [2] http://trochim.human.cornell.edu/kb/stat_t.htm.
- [3] Allen, G., Bengler, W., Dramlitsch, T., 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.
- [4] Allen, G., Bengler, W., Goodale, T., Hege, H.-C., Lanfermann, 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.
- [5] Allen, G., Goodale, T., Lanfermann, G., Radke, T., Seidel, E., Bengler, W., Hege, H.-C., Merzky, A., Masso, J. and Shalf, J., Solving Einstein's Equations on Supercomputers, *IEEE Computer*, 32 (1999) 52-58.
- [6] 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.
- [7] Berman, F., Wolski, R., Figueira, S., Schopf, J. and Shao, G., Application-Level Scheduling on Distributed Heterogeneous Networks. *Supercomputing'96*, 1996.
- [8] Czajkowski, K., Foster, I., Kesselman, C., Sander, V. and Tuecke, S., SNAP: A Protocol for Negotiating Service Level Agreements and Coordinating Resource management in Distributed Systems. *8th Workshop On Job Scheduling Strategies for Parallel Processing*, Edinburgh, Scotland, 2002.
- [9] Dail, H.J., A Modular Framework for Adaptive Scheduling in Grid Application Development Environments. *Computer Science*, University of California, California, San Diego, 2001.
- [10] Dinda, P.A., Online Prediction of the Running Time of Tasks, *Cluster Computing*, 5 (2002).
- [11] Dinda, P.A., A Prediction-based Real-time Scheduling Advisor. *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS 2002)*, 2002.
- [12] 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.
- [13] Dinda, P.A. and O'Hallaron, D.R., Host Load Prediction Using Linear Models, *Cluster Computing*, 3 (2000).
- [14] 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)*, Springer LNCS 1915, Rochester, NY, 2000, pp. 265-280.
- [15] Figueira, S.M. and Berman, F., Mapping Parallel Applications to Distributed Heterogeneous Systems. University of Californian, San Diego, 1996.
- [16] 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.
- [17] Fox, G.C., Williams, R.D. and Messina, P.C., *Parallel Computing Works*, Morgan Kaufmann, 1994, 977 pp.
- [18] 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.

- [19] 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.
- [20] 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.
- [21] 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.
- [22] 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).
- [23] 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.
- [24] Schopf, J.M., A Practical Methodology for Defining Histograms for Predictions and Scheduling. *ParCo'99*, 1999.
- [25] Schopf, J.M. and Berman, F., Stochastic Scheduling. *SuperComputing'99*, Portland, Oregon, 1999.
- [26] 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.
- [27] Turgeon, A., Snell, Q. and Clement, M., Application Placement Using Performance Surface. *HPDC2000*, Pittsburgh, Pennsylvania, 2000.
- [28] 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.
- [29] Weissman, J.B. and Zhao, X., Scheduling Parallel Applications in Distributed Networks, *Journal of Cluster Computing*, 1 (1998) 109-118.
- [30] Wolski, R., Dynamically Forecasting Network Performance Using the Network Weather Service, *Journal of Cluster Computing* (1998).
- [31] 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.
- [32] 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.
- [33] 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.
- [34] 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.