J. Parallel Distrib. Comput. 68 (2008) 1135–1145

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Performance modeling of parallel applications for grid scheduling

H.A. Sanjay, Sathish Vadhiyar*

Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore - 560012, India

ARTICLE INFO

Article history:
Received 27 March 2007
Received in revised form
6 February 2008
Accepted 26 February 2008
Available online 20 March 2008

Keywords:
Performance modeling
Parallel applications
Predictions
Non-dedicated systems
Grids
Cross-platform modeling
Scheduling

ABSTRACT

Grids consist of both dedicated and non-dedicated clusters. For effective mapping of parallel applications on grid resources, a grid metascheduler has to evaluate different sets of resources in terms of predicted execution times for the applications when executed on the sets of resources. In this work, we have developed a comprehensive set of performance modeling strategies for predicting execution times of parallel applications on both dedicated and non-dedicated environments. Our strategies adapt to changing network and CPU loads on the grid resources. We have evaluated our strategies on 8, 16, 24 and 32-node clusters with random loads and load traces from a grid system. Our strategies give less than 30% average percentage prediction errors in all cases, which, to our knowledge, is the best reported for non-dedicated environments. We also found that grid scheduling using predictions of execution times from our performance modeling techniques will lead to perfect mapping of applications to resources in many cases.

1. Introduction

Many computational grid frameworks [35,36,5] are composed of multiple dedicated or non-dedicated clusters, with each cluster consisting of a set of homogeneous machines. Grids have been found to be powerful research-beds for executing various kinds of parallel applications [28,7,3]. Some parallel applications are tightly-coupled involving heavy communications among the parallel tasks [28,3]. These applications exhibit poor performance when executed across multiple clusters due to low-speed network links between the clusters and are typically executed within a single cluster. When a tightly-coupled parallel application is submitted to a grid, a metascheduler evaluates different candidate resource sets, with each candidate set consisting of resources from a cluster, and selects the "most suitable" resources for application execution. The candidate resource sets are mostly evaluated in terms of predicted execution times of the application and the resource set with the minimum predicted execution time is chosen for application execution [28]. Thus, models that predict execution times of the parallel applications on a set of resources are of importance to the efficiency of the scheduling decisions.

Many performance modeling strategies have been proposed in the literature for predicting execution times of parallel applications. However, the existing strategies have different limitations that prevent them from being used for large non-dedicated grid systems.

1. Most of the existing modeling strategies assume uniform loading conditions on the systems when the experiments for modeling are conducted and use the models to predict execution times for large problem sizes and/or larger number of processors for the same loading conditions [39,34,33,6,25,2,1,24,10]. This assumption is unrealistic in non-dedicated environments.
2. Some of the models require source code of the applications for the construction of the models and for instrumentation of the critical components [39,25,2,1,10,41,4]. Analysis and instrumentation of source codes of applications are time-consuming for large complex applications and can prevent large-scale deployment of these applications on grids.
3. Some modeling methods also require analytical models expressing the computation and communication characteristics of the applications [39,25,2,1,34,33,30,29,31]. Building robust analytical models require detailed knowledge of the applications and such knowledge is available only with the application developers. Requiring this knowledge can prevent application developers from integrating their applications into grids.
4. Some of these modeling strategies also perform large number of benchmarks of the individual code segments to automatically determine the analytical models for the code segments [25,2,1,34,33,41,4].
5. Some of the existing efforts for non-dedicated environments can deal with different loading conditions during training the models and predictions, but require the loads to be constant during an

* Corresponding author.
E-mail address: vss@serc.iisc.ernet.in (S. Vadhiyar).

0743-7315/$ – see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2008.02.006

application execution [6,41,4]. Hence these models can deal with grids with only limited amount of non-dedicatedness.

In this work, we have developed a comprehensive set of performance modeling strategies to predict the execution times of tightly-coupled parallel applications on a set of resources in a dedicated or non-dedicated cluster. The main purpose of our prediction strategies is to aid grid metaschedulers in making scheduling decisions. Following are the specific aspects of our performance modeling strategies, based on linear regression, that result in good prediction accuracies of our models on non-dedicated grid systems.

1. Our prediction strategies can deal with non-dedicated systems where the loads can change during application executions. Our techniques periodically monitor and measure loads on the processors and network links during application execution. The aggregates of these measurements across all processors and links are used as parameters of our linear regression models.
2. By continuously updating these aggregate values of loads, and using the predicted values in our performance models, our strategies are able to dynamically adapt to changing CPU and network loads on the grid resources.
3. Our techniques also continuously evaluate the fitness of a performance model function for changing loads and can use different functions at different times for the same application based on grid load and application dynamics.

Besides, our models do not require detailed knowledge and instrumentation of the applications and can be constructed without the involvement of application developers. Moreover, our strategies can derive an initial coarse-level performance model for an application by conducting only up to 30 experiments with the application. The model is subsequently refined with increasing executions of the application by the grid users. These features enable the use of our modeling strategies for rapid and large-scale deployment of parallel applications on non-dedicated grid systems.

We have evaluated our strategies on 4 different clusters for 7 different applications, with both random CPU and network loads and also with load traces obtained from machines in a real grid testbed [18]. We verified our performance models both in terms of average percentage performance prediction errors and also in terms of its usefulness to a metascheduler to arrive at the correct scheduling decisions. Due to the adaptiveness of our models to resource dynamics, we obtained less than 30% average percentage prediction errors in all cases, which to our knowledge, is the best reported for predicting application execution times for any problem sizes and number of processors on non-dedicated systems. We also found that scheduling decisions made with the help of our predictions can result in perfect scheduling in most cases.

In Section 2, we analyze the various existing strategies for performance predictions of parallel applications. In Section 3, we describe in detail our strategy for performance modeling on non-dedicated systems including the various components of our performance models, the use of candidate functions for the models and the cross-platform performance modeling for performance predictions on different clusters. Section 4 details our experiments and results for predicting execution times of different applications on different clusters and with different loading conditions and also compares our strategy with earlier work in terms of modeling overhead. Section 5 describes our current plan of extending our work to deal with large scientific applications. Section 6 gives conclusions of our work and Section 7 presents future work.

2. Related work

Most of the existing efforts on performance modeling deal with dedicated environments and require detailed analytical models and source codes for application components. The work by Xu et al. [39] builds a two-level hierarchical performance model for predicting execution times of parallel programs. At the top level, a graphical model called *thread graph* is constructed for a parallel program. The thread graph, consisting of communication structures, events and segments, represents a high level abstraction of the program and is used by a graph traversal algorithm to estimate the parallel execution time. The execution times of the individual segments of the thread graph are estimated using a low-level model that combines analytical and experimental methods to capture system level effects on performance. However, their model needs analysis of the application for constructing the thread graph for the top level model and analyzing the loop level constructs for the lower level model. The source code also needs instrumentation to obtain the measurements corresponding to small number of iterations in the individual segments to predict the execution times for a larger number of iterations.

The PACE toolkit [25,2] performs analysis of the application's source code to form a set of performance model objects for the application components using CHIPS performance model language. Though PACE can perform predictions on heterogeneous platforms, it needs at least some parts of the source code of the applications and description of the hardware configuration. The POEMS project [1] has built a robust infrastructure including a specification language, component models, a database for storing task dependencies and performance results of individual components, to predict the execution times of the parallel applications. The different components used by the application are specified by a detailed specification language, requiring intervention of the application developer. The POEMS project needs source code of the application for component specification and task graph generation. Prophesy's [34,33] curve-fitting model utilizes both experimental results and the complexities of the applications to predict execution times of applications for larger problem sizes. The curve-fitting models used in Prophesy do not separate system and application parameters and encapsulate all kinds of system dynamics in the model coefficients. The model developer needs detailed knowledge of the components of the application. Secondly, benchmark experiments have to be conducted for various application specific subcomponents including broadcasts, floating-point additions, multiplications etc. The number of such benchmarks increases as more applications are added due to the increase in the number of subcomponents. Our model strategies use only generic benchmarks, that are integral to many grid systems, for measuring CPU and network loads.

Some recent efforts on performance modeling deal with limited kind and amount of non-dedicatedness on the systems. Dimemas [6] is used for analyzing the performance of parallel applications. The parallel application instrumented with instrumentation libraries is executed either on a dedicated or non-dedicated, sequential or parallel system, leading to the generation of Dimemas trace files. These trace files along with the specification of a target parallel system are fed to a simulator which outputs the performance behavior of the application on the target system. Dimemas cannot predict for systems where the loads vary during application execution. The work by Grove et al. [20] requires the application developer to specify the complexity of serial portions and message passing constructs for his application using a performance modeling language. Their work takes into account only the contention caused on the network by the application itself and not due to external load. Our modeling strategies work for non-dedicated environments where the loads on the machines can vary during application execution.

The effort by Yan et al. [41] predicts execution times of parallel applications on non-dedicated heterogeneous systems. They consider a system where a set of workstations can be used for the execution of parallel application and external load can appear in the form of workstation owners executing their own processes. They use Program Execution Graphs (PEGs) for modeling application characteristics. Anglano [4] extended this work by using Petri-Net models for characterizing application behavior. In addition to requiring analysis of the source code and detailed benchmarking, both the efforts use a parameter called O_i that represents the average execution time of the owner process or external load on machine i. Calculation of O_i is not feasible on grid systems where the interference by external load can vary over time. The work by Schopf and Berman [30,29,31] predicts execution times in dynamic non-dedicated environments where the loads can vary at different times. They use stochastic values of load measurements in their component models and obtain stochastic predicted execution times. Their work requires profiling tools for determining the communication and computation requirements of the application. Their component models are also based on detailed parametrized analytical models requiring intervention of the application developer.

3. Methodology

In our modeling method, we calculate the time taken for the execution of parallel application as:

$$T(N, P, \text{minAvgAvailCPU}, \text{minAvgAvailBW}) = \frac{f_{\text{comp}}(N)}{f_{\text{cpu}}(\text{minAvgAvailCPU}) \cdot f_{P\text{comp}}(P)} + \frac{f_{\text{comm}}(N)}{f_{bw}(\text{minAvgAvailBW}) \cdot f_{P\text{comm}}(P)} \quad (1)$$

where

- N: problem size; P: number of processors;
- minAvgAvailCPU, minAvgAvailBW: represent the transient CPU and network characteristics, respectively;
- f_{comp}, f_{comm}: indicate the computational and communication complexity, respectively, of the application in terms of problem size;
- f_{cpu}: function to indicate the effect of processor loads on computations;
- $f_{P\text{comp}}$: used along with computational complexity to indicate the computational speedup or the amount of parallelism in computations;
- f_{bw}: function to indicate the effect of network loads on communications;
- $f_{P\text{comm}}$: used along with communication complexity to indicate the communication speedup or the amount of parallelism in communications.

The formula shown in Eq. (1) splits the execution time of a parallel application into two parts, f_{comp} and f_{comm}, for representing computation and communication aspects, respectively, of the parallel application. This representation is useful for scheduling purposes since a scheduler can allocate the appropriate CPU and network resources of a grid based on the computation and communication requirements, respectively, of the application. The scalability of the computational and communication times with increasing number of processors, is represented by $f_{P\text{comp}}$ and $f_{P\text{comm}}$, respectively. Since, in most parallel applications, the execution times decrease with the increase in number of processors, these functions are contained in the denominators. The increase in CPU and network loads on non-dedicated systems increases the computation and communication times, respectively. minAvgAvailCPU and minAvgAvailBW represent the inverse of the CPU and network loads, respectively. Hence, the corresponding functions, namely, f_{cpu} and f_{bw} are contained in the denominators. Finally, the formula shown in Eq. (1) generalizes the complexity equations of many parallel numerical drivers that deal with memory-resident data [9,17]. Eq. (1) represents a coarse-level model and does not include fine-level performance behavior of the applications including memory access stride and range, cache misses and the corresponding system parameters including cache configurations and memory bandwidth [2,1,11,41]. Our model is primarily intended for scheduling purposes and most of the schedulers of parallel applications [8] do not consider these fine-level parameters.

In order to calculate minAvgAvailCPU and minAvgAvailBW, we measure *AvailCPU* and *AvailBW*. AvailCPU is a fraction of the CPU that can be used for the application and AvailBW of a link is the bandwidth on the link available to an application. Network Weather Service (NWS) [38,37], a tool for forecasting system parameters, was used for obtaining these values. During training the model functions for application, we measure AvailCPUs and AvailBWs on all processors and links involved in application execution at periodic intervals of time from the beginning to the end of the application execution. We then calculate for each processor and link, *AvgAvailCPU* and *AvgAvailBW*, respectively. These are the averages of the periodic AvailCPUs and AvailBWs collected during the application execution. Finally, we calculate *minAvgAvailCPU* and *minAvgAvailBW* values by finding the minimum of *AvgAvailCPU* and *AvgAvailBW* values, respectively, on all processors and links. By considering minAvgAvailCPU and minAvgAvailBW, we assume that the slowest processor and link used by the application affect the overall execution time. Eq. (1) does not consider network latencies since our models are intended for clusters where the nodes are connected by low-latency links.

Our modeling strategy consists of a number of stages for determining the model functions of Eq. (1) consists of a number of stages. Due to uncertainties caused by the non-dedicatedness in the environment, a set of good model functions are chosen at the end of each stage of modeling. For a particular model function, the coefficients are determined by linear regression using the training samples. We evaluate a model function in terms of standard error, SE, defined as: $\text{SSE} = \sum_{i=1}^{N}(y_i - y_i')^2$; $\text{Error\_var} = \frac{\text{SSE}}{(N-p)}$; $SE = \sqrt{\text{Error\_var}}$ where N denotes the number of experiments, y_i, the actual execution time and y_i', the predicted execution time for experiment i, p is the number of terms in the model and SSE is the sum of squares error. The following subsections detail the individual stages.

3.1. Modeling computation

In order to find the computational complexity, f_{comp} in Eq. (1) for the application, we use a candidate set of 77 model functions. These are polynomial, logarithmic, and mixture of polynomial and logarithmic functions and are commonly used in many curve fitting packages and tools [13,12,23]. These functions also encapsulate the behavior of many parallel applications. We execute the application on a single non-dedicated processor with different problem sizes and observe the minAvgAvailCPUs and execution times. Since most of the systems follow round-robin scheduling, we use *minAvgAvailCPU* value for f_{cpu}. For each candidate function, $f_{\text{candidateComp}}$ for f_{comp}, we fit $\frac{f_{\text{candidateComp}}}{\text{minAvgAvailCPU}}$ with the experiment data. At the end of this stage, we sort the functions in terms of the ascending order of their standard error values and choose at most 20 candidate functions for f_{comp} for subsequent stages of modeling.

Table 1
Candidate functions for $f_{P\text{comp}}$ and $f_{P\text{comm}}$

S. No.	$f_{P\text{comp}}(P), f_{P\text{comm}}(P)$	S. No.	$f_{P\text{comp}}, f_{P\text{comm}}$	S. No.	$f_{P\text{comp}}, f_{P\text{comm}}$	S. No.	$f_{P\text{comp}}, f_{P\text{comm}}$
1.	$\sqrt{P}$	5.	$P^{2.5}$	9.	$1/\sqrt{P}$	13.	$1/P^{2.5}$
2.	P	6.	P^3	10.	$1/P$	14.	$1/P^3$
3.	$P^{1.5}$	7.	$\log(P)$	11.	$1/P^{1.5}$	15.	$\frac{1}{\log(P)}$
4.	P^2	8.	$P \cdot \log(P)$	12.	$1/P^2$	16.	$\frac{1}{P \cdot \log(P)}$

Table 2
Candidate functions for f_{bw}

S. No.	f_{bw} (bw)	S. No.	f_{bw}	S. No.	f_{bw}	S. No.	f_{bw}
1.	$\sqrt{bw}$	3.	$bw^{1.5}$	5.	$bw^{2.5}$	7.	$\log(bw)$
2.	bw	4.	bw^2	6.	bw^3	8.	$bw \cdot \log(bw)$

3.2. Modeling communication

To find the communication complexity, f_{comm}, and the effect of network loads on the application, represented by f_{bw} in Eq. (1), we execute the parallel application on 2 processors with different problem sizes and under different CPU and network loads. For each of the experiments, we observe minAvgAvailCPU and minAvgAvailBW values. We then use the following equation for execution times on 2 processors:

$$T1(N, \text{minAvgAvailCPU}, \text{minAvgAvailBW}) = \frac{f_{\text{comp}}(N)}{2 \cdot \text{minAvgAvailCPU}} + \frac{f_{\text{comm}}(N)}{2 \cdot f_{bw}(\text{minAvgAvailBW})}. \quad (2)$$

For f_{comp} in Eq. (2), we use the 20 functions with top accuracy values determined in the computation modeling phase. For f_{comm}, we use the 77 candidate functions used for computation modeling. For f_{bw}, we evaluate 8 candidate functions shown in Table 2. Thus, we evaluate a total of 12 320 ($20 \times 77 \times 8$) combinations of functions for Eq. (2) in terms of standard error values and fit the execution times T1 with the actual execution times. For each of the 20 functions for f_{comp}, we choose at most 10 combinations of functions for f_{comm} and f_{bw} with minimum standard error values to form *filtered_list*. Finally, 20 combinations of f_{comp}, f_{comm} and f_{bw} with minimum standard error values are chosen from the *filtered_list* for the next stage.

3.3. Modeling scalability

In the final stage, the scalability functions, $f_{P\text{comp}}$ and $f_{P\text{comm}}$ of Eq. (1) are determined. The application is executed with 2, 4 and 8 processors under different loading conditions and the resulting execution times are observed. We then fit different combinations of functions in Eq. (1) with the observed execution times. For each of the 20 f_{comp}, f_{comm} and f_{bw} combinations determined at the end of the communication modeling phase, we evaluate different combinations of functions for $f_{P\text{comp}}$ and $f_{P\text{comm}}$. The 16 candidate functions for $f_{P\text{comp}}$ and $f_{P\text{comm}}$ are shown in Table 1. Thus, we evaluate a total of 5120 ($20 \times 16 \times 16$) combinations of functions in this stage. For each of the 20 f_{comp}, f_{comm} and f_{bw} combinations, we choose at most 50 combinations of $f_{P\text{comp}}$ and $f_{P\text{comm}}$ functions with minimum standard error values. Thus, at the end of the training phase, we obtain a list of at most 1000 combinations of functions for Eq. (1). The combinations are sorted in the ascending order of their standard error values to form a *sorted_list*.

While choosing the top functions in each stage, we ensure that the chosen functions have standard errors that are at most 20% greater than the minimum standard error and that the number of chosen functions are below a threshold limit. The values for these limits were found by trial-and-error in order to provide flexibility for our prediction strategies to dynamically select, based on resource dynamics, one of the functions with minimum standard error and, yet limit the number of function evaluations for a given prediction. Some parallel applications may have constraints regarding the minimum number of processors that they need to execute. In these cases, we cannot use single processor executions to determine the computation complexity functions. We first execute the application with different problem sizes on the minimum number of processors. We then form different combinations of $\frac{f_{\text{comp}}}{\text{minAvgAvailCPU}}$, f_{comm} and f_{bw} functions for Eq. (1) and fit the model shown in the equation with the actual execution times. To prune down the number of function evaluations, we follow a procedure that eliminates candidate computation and communication functions which lead to large standard error values for greater than a specified number of times. We then proceed to the scalability modeling phase.

3.4. Prediction of execution times

To predict the execution time of an application with a given problem size and given number of processors, the top combination of functions in the sorted_list, obtained in the training phase, is used in Eq. (1). The prediction of execution time needs values for problem size, number of processors, minAvgAvailCPU and minAvgAvailBW. Problem size and number of processors for execution are obtained from the user. During training the models, the minAvgAvailCPU and minAvgAvailBW values were obtained by observing the system loads during application execution. For prediction of execution time of an application, these values have to be predicted, since the values represent the system loads that will exist during the period of application execution. Hence, we forecast the values based on the history of load dynamics measured on the system using the forecasting tools from NWS.

We then use the problem size and number of processors supplied by the user and the predicted MinAvgAvailCPU and MinAvgAvailBW to predict the execution time of the application. After the application is executed on the resources chosen by a grid scheduler, the actual execution time of the application and the MinAvgAvailCPU and MinAvgAvailBW values are observed and added to the training data along with the problem parameters as a data point. The combinations of functions in the sorted_list are evaluated by means of standard error values with the updated training data. This can lead to reordering of the combinations in the sorted_list and use of a different combination for the next prediction. In order to avoid evaluating all the combinations in the sorted_list at the end of every prediction, we eliminate those combinations of functions whose ranks in the sorted_list are within the lowest 10 percentile for 5 continuous predictions. Thus the sorted_list shrinks over time and converges to contain only the most promising combinations. In order to take into account new load dynamics, the original set of candidate combinations of functions are reconsidered by revisiting the communication modeling phase of training at the end of every 50 predictions.

3.5. Cross-platform performance modeling

We have also developed techniques whereby the sorted list of model functions and training data obtained for an application on a given parallel platform called *reference platform* can be used for predicting execution times of the application on another parallel platform called *target platform*. In order to use a combination of functions obtained on a *reference platform* for a *target platform*, the coefficients of the different functions of Eq. (1) have to be scaled to take into account the performance difference between the reference and the target platforms. The computation complexity f_{comp} is scaled by a CPU scaling factor, *Rcpu*, which is determined by obtaining single processor execution times of the application with a moderate problem size on dedicated target and reference processors, and finding the ratio between the two times. In order to determine the scaling factor for the communication complexity, we conduct dedicated 2-processor experiments with different problem sizes on the reference and target platforms. The top model in the sorted_list is then fitted with the 2-processor experiment data on the reference platform. The coefficients of the functions in the top model and the resulting goodness of fit are determined. The trained function is then fitted with the 2-processor experiment data on the target platform after scaling the computation complexity with *Rcpu*. The communication complexity of the resulting function is then scaled with different scaling factors until the goodness of fit for the 2-processor experiment data on the target platform matches with the goodness of fit obtained on the reference platform. The resulting scaling factor is noted as the bandwidth scaling factor, *Rbw*. *Rbw* may be a non-unit real number due to difference in communication and computation overlaps in different platforms.

For predicting the execution time of an application on a target platform, the top model in the sorted_list with the minimum standard error, obtained on the reference platform, is considered. The coefficients of the functions in the model are obtained by training the model with the data corresponding to non-dedicated executions for different problem sizes and processors available for the reference platform. If $f_{\text{comp}}, f_{\text{comm}}, f_{P\text{comp}}, f_{P\text{comm}}, f_{\text{cpu}}, f_{bw}$ are the functions for the top model obtained on the reference platform, the predicted execution time of the application with problem size N and P processors of the target platform is then given by Eq. (3). The minAvgAvailCPU and minAvgAvailBW values shown in the equation are obtained by forecast of CPU and network loads on the target platform.

$$T_{\text{predicted}} = \frac{Rcpu \cdot f_{\text{comp}}(N)}{f_{\text{cpu}}(\text{minAvgAvailCPU}) \cdot f_{P\text{comp}}(P)} + \frac{Rbw \cdot f_{\text{comm}}(N)}{f_{bw}(\text{minAvgAvailBW}) \cdot f_{P\text{comm}}(P)}. \quad (3)$$

4. Experiments and results

4.1. Experiment setup

Experiments were conducted validating the modeling strategies in terms of predictions of execution times. In order to avoid intrusion on production grid systems, we conducted experiments on systems that are under our administrative control. The experiments were conducted on 4 different clusters:

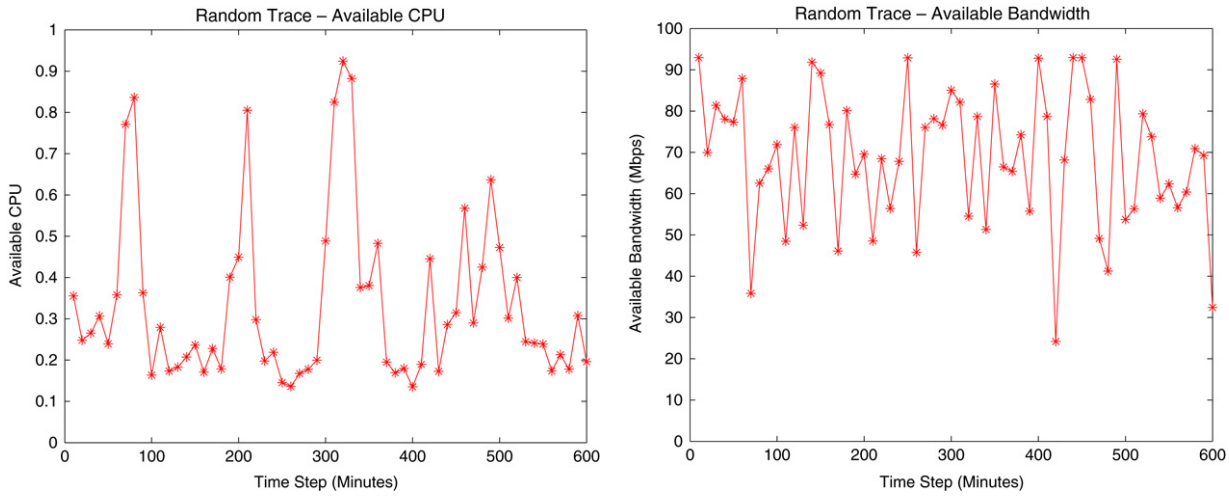
1. a **8-processor Intel Pentium IV cluster** with each processor having 2.8 GHz speed, Fedora Core 2.0, Linux 2.6.5–1.358 operating system, a 512 MB RAM, a 80 GB hard disk, and connected by a 100 Mbps switched Ethernet;
2. a **32-processor IBM P720 cluster** arranged in 8 nodes, with each node a 4-way IBM Power 5 SMP with hard disk capacity of 146 GB and running Suse Linux 9.0 sp 1 operating system, each processor having 1.65 GHz CPU speed and 1 GB RAM and all the 8 nodes are connected to each other by Gigabit Ethernet links through Gigabit Nortel switch;
3. a **16-processor AMD cluster** arranged as 8 dual-core AMD Opteron 1214 based 2.21 GHz Sun Fire servers running CentOS release 4.3 with 2 GB RAM, 250 GB Hard Drive and connected by Gigabit Ethernet and
4. a **24-processor Woodcrest cluster** arranged as 12 Dual-Core Intel Xeon 5130 based 2.0 GHz HP xw6400 Workstations running Fedora Core 6 (Zen), with 4 GB RAM, 160 GB Hard Drive and connected by 100 Mbps Ethernet.

The IBM cluster was used in a dedicated mode with the help of space-sharing using IBM LoadLeveler batch queuing system. The other clusters were used in non-dedicated modes. On the dual-core AMD and Woodcrest systems, *taskset* option was used to control the process assignment to the cores. On all the clusters, 1, 2, 4 and 8 processors were used for training the models and up to the maximum number of available processors were used for predictions. During training, about 30 problem sizes within a small problem size range were used and during predictions, random problem sizes within and outside the training problem size range were used.

We have tested our prediction strategies with 7 different parallel applications.

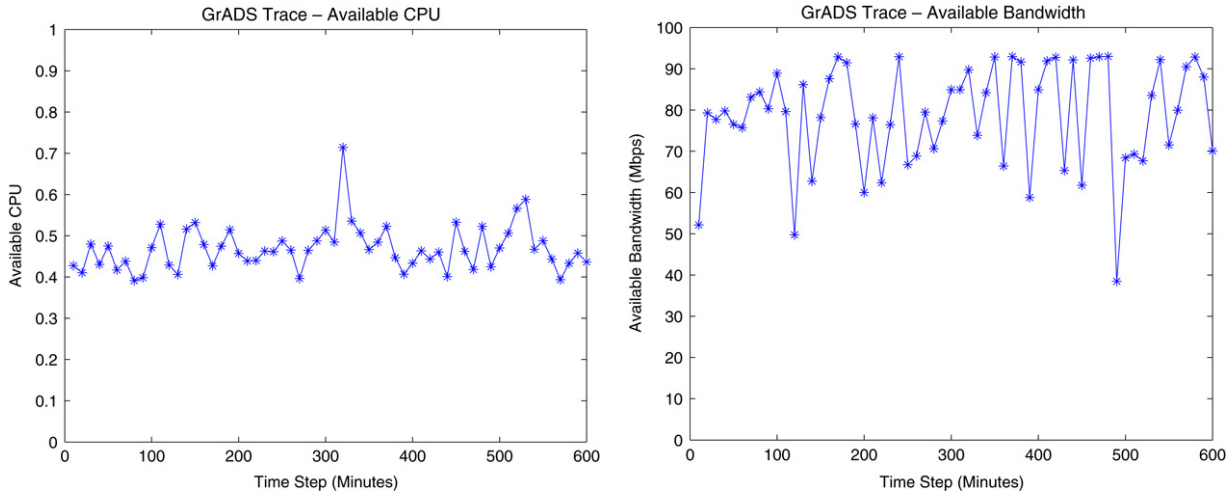
1. ScaLAPACK [9] eigen value solver for a double precision symmetric matrix using PDSYEV kernel. 2-D block cyclic distribution was used.
2. 1-D FFT application from FFTW [17] package. The transform data are distributed over multiple processes. The application uses routines for parallel one-dimensional transforms of complex data.
3. Conjugate Gradient (CG) application to solve a system of linear equations with a real symmetric positive definite matrix. Diagonal matrix is used for preconditioning. Row-wise block striped partitioning was used.
4. Molecular dynamics simulation (MD) of Lennard-Jones system systolic algorithm. N particles are divided evenly among the P processes running on the parallel machine. The calculation of forces is divided into P stages. The traveling particles are shifted to the right neighbor processor in a ring topology.
5. Poisson Solver using 2-D Jacobi. Uses 1-D domain decomposition with non-blocking communication. The domain decomposition is along the x-axis. The application is run until the maximum number of Jacobi iterations are reached.
6. Integer Sort (IS). Parallel Integer sort application. Local sorting by individual processes followed by a global sort across processes.
7. Symmetric Successive Over-Relaxation (SSOR) using red-black ordering. Cartesian topology is used for arrangement of processes. Grid points are updated using five-point finite-difference stencil.

During the course of an application execution, available CPUs of the nodes and available bandwidths of the inter-node links are collected every 2 min. We used synthetic loads on our system to simulate the load conditions on real grid systems. 2 different loading conditions were used for our experiments: *random* and *grads*. In random loading, synthetic CPU and network loading programs were continuously run on the processors in the background. For loading the CPUs of a system at a given point of time, a set of processors was randomly chosen, random amounts of loads were introduced on each of the processors and the loads were maintained for random amounts of time. For network loading, we used a loading program to introduce synthetic network loads on the links of the system and to reduce the available bandwidths of the links. At a given point of time, a random number of source-destination pairs is chosen, and random amounts of network loads are introduced on the links between the source-destination pairs, and the loads are maintained for random durations. This process of



(a) Available CPUs. (b) Available bandwidths.

Fig. 1. Available CPUs and bandwidths for random loading conditions.



(a) Available CPUs. (b) Available bandwidths.

Fig. 2. Available CPUs and bandwidths for grads loading conditions.

random CPU and network loading is repeated continuously on the system. The details regarding the amount and duration of loads in random loading can be found in our earlier work [40]. In the grads loading, the traces [19] of CPU and network loads measured by NWS in the GrADS [18] grid research-bed were used to guide the parameters to our synthetic CPU and network loading programs. These parameters include the amount and duration of each CPU and network loading, on each processor and link, respectively. The load traces on nodes of the GrADS research-bed, that had similar CPU and network characteristics to the nodes of our clusters, were used for loading. Even though we use simulated load conditions of GrADS testbed, we verified that the load dynamics introduced by our simulated loads matched with those on the machines of the GrADS testbed [19].

Figs. 1 and 2 show the available CPU and available bandwidth values for a 3 h period on a processor and link, respectively, in the Intel system, for random and grads loading conditions, respectively. From the figures, we find that the CPU loads on the GrADS real grid testbed are more stable than the random CPU loads. However, the large variations in the random CPU loads help to represent systems with high load dynamics and hence help in better validation of the robustness of our performance modeling strategies than the grads CPU loads. We find that the random network loads have similar dynamics as the grads loads and hence are realistic. Thus, by conducting experiments both on extreme load dynamics and real grid conditions, we show the usefulness of our models under different conditions.

For evaluating the modeling strategies, we calculated the average percentage prediction errors, and also determined the usefulness of the strategies for scheduling. Since our modeling strategies are primarily meant to improve the quality of scheduling in grid systems, a more useful evaluation related to scheduling is to compare the minimum execution times predicted by our model and the minimum actual execution times for various problem sizes. We first split the problem sizes of an application into different groups such that the actual execution times of the application when executed with different problem sizes in a group on a dedicated system differ by a maximum of 50 s. For each of the problem size groups, we obtain actual and predicted execution times for different problem sizes in the group and different number of processors. We then obtain the problem size, *minPredictedProblemSize*, and number of processors, *minPredictedProcessors*, corresponding to minimum of the predicted execution times. We then obtain the actual execution time, *actualForminPredicted*, corresponding to *minPredictedProblemSize* and *minPredictedProcessors*. We compare

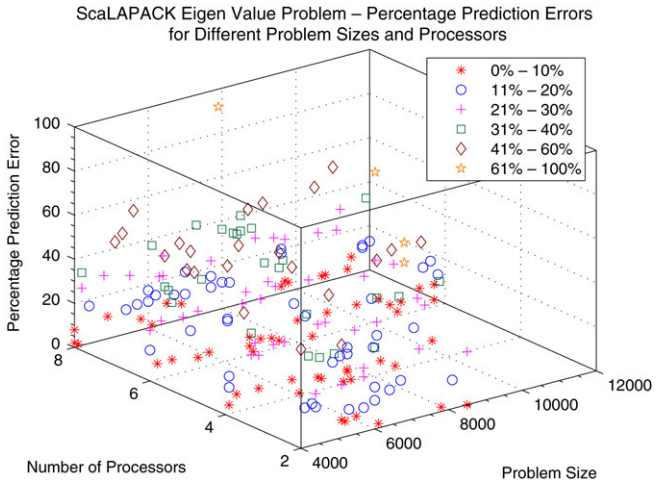


Fig. 3. Percentage prediction errors (PPE) for ScaLAPACK eigen value problem on the intel cluster with random loading. Different colored data points correspond to different ranges of percentage prediction errors. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

actualForminPredicted with *minActual*, the minimum of actual execution times. The percentage difference between the two values indicate the loss in efficiency a grid scheduler would incur if it uses the predictions by our modeling strategies for evaluating the candidate schedules. For a perfect scheduler, the percentage difference should be 0.

4.2. Results

Fig. 3 plots the percentage prediction errors for different problem sizes and number of processors for the eigen value problem. We find that the percentage prediction errors are less than 30% for 72% of the predictions and less than 40% for 85% of the predictions. Fig. 4 plots the percentage prediction error values for different experiments in the order the experiments were conducted for the eigen value problem. We find that during the initial experiments, the error values are high with some predictions having about 40–85% percentage prediction errors. In the latter stages, the error values converge to 20–30%. The figure also shows predictions by a single model for all the experiments. We chose the model that was the best or had minimum standard error at the end of the training phase for these predictions.

Table 3
Predictions on the 8-processor Intel cluster

Problem	Load type	PPE (single model)		PPE (multiple models)	
		Avg.	Std. Dev.	Avg.	Std. Dev.
Eigen	Random	25.41	20.79	22.47	16.73
Eigen	Grads	40.5	37.2	27.93	21.2
FFT	Random	31.16	26.9	25.16	22.63
CG	Random	22.2	16.22	20.82	15.79
MD	Random	18.85	13.66	16.66	14.30
MD	Grads	10.05	7.47	8.7	7.67
Poisson	Random	31.62	24.96	27.86	22.36
IS	Random	11.53	9.01	11.68	9.38
IS	Grads	11.00	14.8	11.4	15.59
SSOR	Random	22.98	19.65	16.22	13.78
SSOR	Grads	10.5	11.8	9.69	8.07

Table 4
Predictions on 16-processor AMD and 24-processor Woodcrest (WC)

Prob. : Cluster	Load	PPE (single model)		PPE (multiple models)	
		Avg.	Std. Dev.	Avg.	Std. Dev.
Eigen:AMD	Random	34.7	23.2	23.3	19.6
Eigen:AMD	Grads	17.38	14.96	16.61	11.30
MD:AMD	Random	23.7	19.7	25.05	20.2
MD:AMD	Grads	12.06	11.08	13.09	13.34
Poisson:AMD	Grads	14.47	12.96	15.95	13.88
SSOR:AMD	Grads	35.25	83.93	13.49	11.07
IS:AMD	Grads	32.02	57.89	13.88	16.46
Poisson:WC	Random	37.41	19.53	28.78	14.37
IS:WC	Random	38.24	23.73	27.7	20.93

As can be seen, using a mixture of good models and choosing different models at different points of time give smaller prediction errors than using a single model for predictions on non-dedicated environments. The average percentage prediction errors for all experiments using multiple models is 22.47% with a standard deviation of 16.73% while the average using a single model is 25.41% with a standard deviation of 20.79%.

Tables 3 and 4 summarize the results regarding percentage prediction errors obtained on the 8-processor Intel, 16-processor AMD and 24-processor Woodcrest clusters. The results demonstrate that our modeling strategies give good predictions for both random loading conditions and the loading conditions that exist on one of the current grid systems, and for higher number of processors than those used during the training.

Table 5 indicates the usefulness of prediction methodology for scheduling a MD application on the non-dedicated Intel cluster

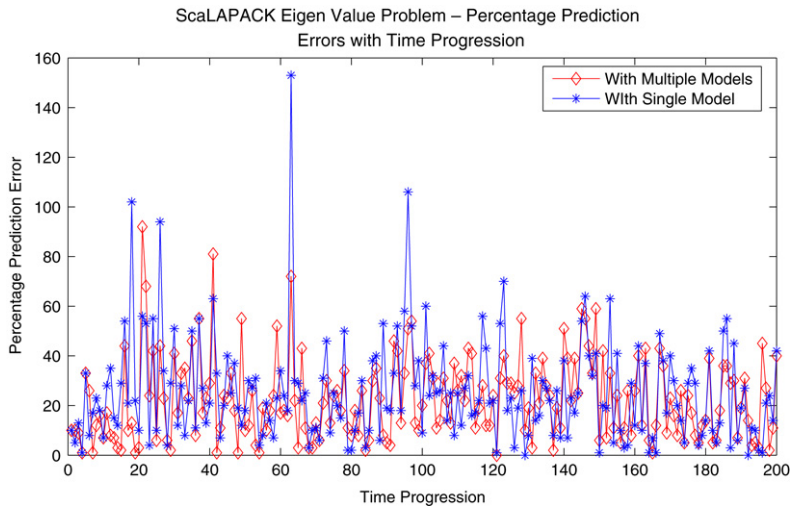


Fig. 4. Percentage prediction errors (PPE) at different times for ScaLAPACK eigen value problem on the Intel cluster with random loading.

Table 5
Usefulness of predictions for scheduling of MD on the 8-processor Intel cluster with random loading

Problem size group	*N* and *P* for minimum predicted execution time: (1)	*N* and *P* for minimum actual execution time: (2)	Actual exec. time for (1): (3)	Actual exec. time for (2): (4)	% Increase in exec. time due to prediction ((3 − 4)/4)
864–936	864,8	864,8	428.92	428.92	0.00
1032–1080	1032,8	1032,8	589.06	589.06	0.00
1296–1344	1296,8	1296,8	825.17	825.17	0.00
1440–1488	1440,8	1440,8	927.55	927.55	0.00
1608–1656	1608,8	1608,8	1166.04	1166.04	0.00
1704–1776	1704,8	1704,8	1292.47	1292.47	0.00
1896–1944	1944,8	1944,8	1650.34	1650.34	0.00
2256–2304	2280,8	2304,8	2637.62	2439.54	8.00
2400–2448	2400,8	2448,8	2896.47	2757.99	2.00
2568–2616	2592,8	2592,8	3068.87	3068.87	0.00

with random loading. The last column denotes the percentage difference between *actualForminPredicted* and *minActual* and is indicative of the loss in efficiency of a scheduler when using the predicted execution times by our modeling strategies. As can be seen, the maximum percentage increase in execution time when a scheduler uses our predicted times is only 8%. In a grid system with high load dynamics, where it is difficult to achieve perfect scheduling, the loss in efficiency of the scheduler by only 8% is tolerable. We also find that except for 2 problem size groups, using our performance modeling strategies to schedule molecular dynamics application on the resources will give rise to perfect scheduling as can be seen by the 0 values in the last column. We obtained similar good results for scheduling for ScaLAPACK eigen value problem, integer sort and SSOR applications.

4.3. Modeling overheads

The following lists the various overheads in deriving a performance model using some of the techniques which are representative of the other performance modeling strategies described earlier:

1. Adve and Vernon [1]: (Overhead for deriving a deterministic graph for the application) + ([number of problem size configurations] × [number of processor configurations] × [number of tasks in the application]) experiments for model derivation.
2. Prophesy [34,33]: ([number of problem size configurations] × [number of processor configurations]) experiments for model derivation.
3. Anglano [4]: (Overhead for source code analysis for identifying computation and communication segments) + ([Number of segments] × [number of problem size configurations] × [number of processor configurations]) experiments for model derivation.
4. Lee et al. [24]: (300–1000) experiments for model derivation.
5. PACE [25,2]: (Overhead for source code analysis to identify subtasks) + (Overhead for mapping their performance modeling language outputs to the subtasks for compilation).
6. Ipek et al. [21]: (250–500) experiments for model derivation.
7. Our strategy: (20 1-processor + 20 2-processor + 10 4-processor + 10 8-processor) experiments for model derivation + (overhead for deriving sorted list of functions [∼5 min]).

We find that some of the strategies require user intervention to manually analyze the source code of the application and the associated overhead cost is highly non-deterministic. The number of experiments needed to derive models in some strategies are functions of number of processor and problem size configurations. Our modeling strategy does not need source code analysis by the user and need the smallest number of experiments to derive the models. The time taken for a prediction with our model is 2–4 s. The primary reason for the small number of experiments in our strategy is that we do not require our initial models to be accurate. Rather, we continuously evaluate our models as applications are executed in grid systems.

In terms of implementation in production grid systems, all modeling strategies require the application developer to register information about the locations of the executables to a centralized application database. Models requiring source codes [39,25,2,1,10,41,4,31] incur additional registration overhead for obtaining the source files and other libraries for the application. These models also incur the overhead of extensive analysis and instrumentation of the source code. The existing models for dedicated systems [39,34,6,25,2,1,24,10] incur synchronization overheads among the processors to ensure dedicatedness during experiments for deriving the model functions. Our modeling strategy incurs the following additional overheads:

1. During the experiments, our strategy incurs the overhead of monitoring the cpu and bandwidth loads. However, many of the grid systems consist of infrastructures for monitoring information about resources [38].
2. Our models also incur a small overhead in porting the functions derived on one cluster to other clusters for cross-platform modeling. However, while the existing strategies perform extensive experiments on all clusters to derive model functions for each cluster, our strategy, due to cross-platform modeling, requires extensive experiments on only one cluster.
3. After an application is executed on a cluster, our strategy incurs the overhead of adding parameters corresponding to the application run to the cluster database and reevaluation of the model functions. This step is needed for model adaptivity to changing load dynamics. Moreover, these evaluations are conducted simultaneously with the predictions, i.e. a scheduler can make use of a previous model for immediate predictions while the model is refined for use in later predictions.

4.4. Cross-platform performance predictions

For validating the cross-platform performance prediction techniques described in Section 3.5, we used 3 platforms: the 8-processor Intel cluster, 16-processor AMD cluster and 32-processor IBM cluster. For a given pair of reference and target clusters and for a given application, we use the top model function for the application that was obtained after conducting *exp-refer* experiments, corresponding to different problem sizes and processors, on the reference cluster. The computation and communication functions of the model are then scaled by the corresponding scaling factors, as explained in Section 3.5, and the resulting model is then used to predict execution times for the application on the target cluster. The percentage prediction errors for the model on the target cluster were obtained by conducting *exp-target* experiments with the application, corresponding to different problem sizes and processors, on the target cluster and obtaining the actual and predicted execution times for the experiments.

Table 6
Cross-platform predictions on the 8-processor intel, 16-processor AMD and 32-processor IBM clusters

Prob.	Reference cluster	Loading on reference cluster	exp-refer	Target cluster	Loading on target cluster	exp-target	Avg. PPE
Eigen	Intel	Random	309	IBM	Dedicated	50	18.53
Eigen	AMD	Grads	203	IBM	Dedicated	50	25.9
Eigen	Intel	Random	309	AMD	Grads	150	26.67
MD	Intel	Grads	212	IBM	Dedicated	50	23.04
MD	AMD	Grads	206	IBM	Dedicated	50	22.08
MD	Intel	Grads	212	AMD	Random	100	28.23
MD	AMD	Grads	206	Intel	Grads	150	18.68
SSOR	Intel	Grads	235	AMD	Grads	150	21.20
SSOR	AMD	Grads	240	Intel	Grads	150	22.55
IS	Intel	Random	300	IBM	Dedicated	100	25.70
IS	Intel	Random	300	AMD	Grads	150	26.06
IS	AMD	grads	226	Intel	Grads	123	27.08

Since the IBM cluster is a space-shared batch system, the processors are not subject to external load and are dedicated for our experiments. Hence, we used a value of 1 for minAvgAvailCPU, i.e. unloaded processor, for our experiments on the IBM cluster. However, the network links can still be subjected to external load since the other user applications may cause network traffic on the shared switches and links used by our experiments. We measured the bandwidths on a link of the cluster periodically for 4 days and used the average of the bandwidths for minAvgAvailBW.

Table 6 summarizes the prediction results due to cross-platform modeling for different reference and target cluster combinations. About 60–100% of predictions were obtained with less than 30% prediction error. The average percentage prediction error is less than 30% in all cases. We also find that model functions and training data obtained on a non-dedicated reference platform can be used for good predictions on other non-dedicated environments under different loading conditions and dedicated batch systems.

4.5. Comparison with a modeling strategy for non-dedicated systems

We compared the predictions by our modeling strategies with the modeling strategy by Schopf et al. [30,29,31] for non-dedicated systems. We considered the same 2D SSOR problem that they had considered in their work. Stochastic values of available bandwidths and available CPUs based on the NWS traces were input to the Schopf's model for predicting execution times. We found that the average percentage prediction errors due to our modeling were less than 20% while the average percentage prediction errors due to Schopf' modeling were between 27–45%. We also found that for large problem sizes, the stochastic values by Schopf have large ranges and hence will not be useful for grid schedulers to make scheduling decisions.

4.6. Summary

In summary, in all our experiments, 48–98% of predictions were obtained with less than 30% prediction error. The average percentage prediction error is less than 30% in all cases. Although these percentages are high in the context of predictions on unloaded environments, these numbers are reasonable and to our knowledge, the best reported on non-dedicated environments. Using multiple models was found to be beneficial over using a single model for prediction in many cases. Also, using single model for predictions resulted in greater than 30% average percentage prediction errors in some cases. Thus, our approach of adaptively using different models for different predictions yield good prediction results in non-dedicated systems. We also showed that scheduling using our predictions will result in perfect scheduling in many cases and the maximum loss in efficiency of a scheduler when using our models is only 11%. Our modeling overheads are smaller and more deterministic when compared to other models. Our cross-platform techniques were able to give less than 30% average percentage predictions when porting the models to either dedicated or non-dedicated platforms.

5. Performance predictions for large scientific applications

The performance modeling techniques discussed in this work are intended for simple parallel kernels that have single phase of uniform computations and communications among processors. We plan to extend this work for large scientific applications where the computation and communication complexities can vary in different phases of application execution and where the amount of computations and communications within a phase can be non-uniform among different processors [26,15,22].

For predicting the execution times of large applications with multiple phases of computation and communication complexities, we plan to investigate the appropriateness of various available techniques for phase detection and prediction [14,16,32,27]. These techniques use various application parameters including working sets, conditional branches and basic blocks to identify phases in the application. The primary challenge will be to study the usefulness of the techniques for various kinds of non-dedicatedness of the systems. Another challenge is to determine the appropriate thresholds in variation of performance metrics that can be used to define phase boundaries. For each of the detected phases, we can then use the CPU and network load measurements and execution times within the phase boundaries to derive per-phase execution models as shown in Eq. (1). The predictions by the per-phase execution time models can then be used by rescheduling strategies to dynamically migrate the application to different sets of resources suitable for a phase as the application enters the phase.

Our work can be easily extended to deal with non-uniform computations and communications among processors within a single phase of application execution. In this case, instead of modeling the execution time for the entire application, we model per-process execution times as shown in Eq. (4).

$$T_{\text{pid}}(N, P, \text{AvgAvailCPU}_{\text{pid}}, \text{AvgAvailBW}_{\text{pid}}) = \frac{f_{\text{comp-pid}}(N)}{f_{\text{cpu-pid}}(\text{AvgAvailCPU}_{\text{pid}}) \cdot f_{\text{Pcomp-pid}}(P)} + \frac{f_{\text{comm-pid}}(N)}{f_{\text{bw-pid}}(\text{AvgAvailBW}_{\text{pid}}) \cdot f_{\text{Pcomm-pid}}(P)}. \quad (4)$$

In Eq. (4), T_{pid} is the time for executing the phase by a process with identifier, *pid*, and can be different for different processes. While this execution time is the time between beginning and end of the phase for some processes, it is the time between beginning of the phase and the beginning of large wait times spent in synchronizations at the end the phase for the other processes. $\text{AvgAvailCPU}_{\text{pid}}$ and $\text{AvgAvailBW}_{\text{pid}}$ represent the average CPU and network loads, respectively, during the phase and correspond

to the processor on which the process, *pid* is executed. $f_{\text{comp-pid}}$, $f_{\text{comm-pid}}$, $f_{\text{cpu-pid}}$, $f_{\text{bw-pid}}$, $f_{\text{Pcomp-pid}}$ and $f_{\text{Pcomm-pid}}$ are the functions for process, *pid*, and correspond to the functions shown in Eq. (1) for the entire application.

6. Conclusions

In this work, we had devised performance modeling techniques for predicting execution times of tightly-coupled parallel applications for the purpose of scheduling the applications on grid resources. We have also developed cross-platform modeling techniques for porting the results of performance modeling on one platform or cluster to other clusters in the grid. The overheads of our modeling strategies were shown to be less than that of the other techniques. Due to the consideration of resource dynamics during application executions and dynamic evaluation of performance model functions, our performance modeling strategies gave less than 30% average percentage prediction errors in all cases, which is reasonable for non-dedicated systems. We also found that scheduling based on predictions by our strategies will result in perfect scheduling in many cases.

7. Future work

Our future work is to develop robust grid scheduling techniques that efficiently use the predictions from our performance models. We also plan to augment our techniques for predicting execution times for complex multi-phase and multi-component applications where the computation and communication complexities can drastically change between different phases of application execution. The observations about the multiple phases will be used to build efficient rescheduling techniques for migrating the applications to different resources at the phase boundaries. We plan to extend our modeling techniques to model the I/O costs in the applications.

References

[1] V. Adve, M. Vernon, Parallel program performance prediction using deterministic task graph analysis, ACM Transactions on Computer Systems 22 (1) (2004) 94–136.

[2] A.M. Alkindi, D.J. Kerbyson, G.R. Nudd, Dynamic instrumentation and performance prediction of application execution, In High Performance Computing and Networking (HPCN2001) 2110 (2001) 313–323.

[3] G. Allen, T. Dramlitsch, I. Foster, N. Karonis, M. Ripeanu, E. Seidel, B. Toonen, Supporting efficient execution in heterogeneous distributed computing environments with cactus and globus, in: Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM), 2001.

[4] C. Anglano, Predicting parallel applications performance on non-dedicated cluster platforms, in: ICS '98: Proceedings of the 12th international conference on Supercomputing, 1998.

[5] ApGrid - Asia Pacific Grid, http://apgrid.org.

[6] R. Badia, J. Labarta, J. Gimenez, F. Escale, DIMEMAS: Predicting MPI applications behavior in grid enviornaments., in: In Workshop on Grid Applications and Programming Tools (GGF8), Seattle York, U.S.A, 2003.

[7] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, L. Marchal, Y. Robert, Centralized versus distributed schedulers for multiple bag-of-task applications, in: 20th International Parallel and Distributed Processing Symposium, 2006.

[8] F. Berman, R. Wolski, The AppLeS Project: A Status Report, in: Proceedings of the 8th NEC Research Symposium.

[9] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R.C. Whaley, ScaLAPACK Users' Guide, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[10] R. Block, S. Sarukkai, P. Mehra, Automated performance prediction of message-passing parallel programs, in: Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM), 1995.

[11] L. Carrington, A. Snavely, N. Wolter, A performance prediction framework for scientific applications, Future Generation Computer Systems 22 (3) (2006) 336–346.

[12] CurveExpert, http://curveexpert.webhop.biz.

[13] DataFit, http://www.curvefitting.com.

[14] A. Dhodapkar, J. Smith, Comparing program phase detection techniques, in: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003.

[15] W. Dick, M. Heath, Whole system simulation of solid propellant rockets, in: Proceedings of the 38th Joint Propulsion Conference and Exhibit, Indianapolis, IN, USA, 2002.

[16] C. Ding, S. Dwarkadas, M. Huang, K. Shen, J. Carter, Program Phase Detection and Exploitation, in: 20th International Parallel and Distributed Processing Symposium, 2006.

[17] M. Frigo, S. Johnson, The design and implementation of FFTW3, Proceedings of the IEEE 93 (2) (2005) 216–231, special issue on Program Generation, Optimization, and Platform Adaptation.

[18] The GrADS Project, http://www.hipersoft.rice.edu/grads.

[19] GrADS Traces, http://pompone.cs.ucsb.edu/rich/data.

[20] D.A. Grove, P.D. Coddington, Modeling message-passing programs with a performance evaluating virtual parallel machine, Performance Evaluation 60 (1-4) (2005) 165–187.

[21] E. Ipek, B. de Supinski, M. Schulz, S. McKee, An Approach to Performance Prediction for Parallel Applications, in: Euro-Par, vol. 3648, Springer LNCS, 2005.

[22] D. Kerbyson, H. Alme, A. Hoisie, F. Petrini, H. Wasserman, M. Gittings, Predictive performance and scalability modeling of a large-scale application, in: Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM), Denver, Colorado, USA, 2001.

[23] LabFit, http://www.angelfire.com/rnb/labfit.

[24] B. Lee, D. Brooks, B. de Supinski, M. Schulz, K. Singh, S. McKee, Methods of inference and learning for performance modeling of parallel applications, in: Proceedings of Symposium on Principles and Practice of Parallel Programming (PPoPP'07), San Jose, California, U.S.A, 2007.

[25] G. Nudd, D. Kerbysin, E. Papaefstathiou, S. Perry, J. Harper, D. Wilcox, PACE - a toolset for the performance prediction of parallel and distributed systems, The International Journal of High Performance Computing Applications 14 (3) (2000) 228–251.

[26] L. Oliker, R. Biswas, H. Shan, J. Singh, Design strategies for irregularly adapting parallel applications, in: Proceedings of the Tenth SIAM Conference on Parallel Processing for Scientific Computing, 2001.

[27] E. Perelman, M. Polito, J.-Y. Bouguet, J. Sampson, B. Calder, C. Dulong, Detecting phases in parallel applications on shared memory architectures, in: 20th International Parallel and Distributed Processing Symposium, 2006.

[28] A. Petitet, S. Blackford, J. Dongarra, B. Ellis, G. Fagg, K. Roche, S. Vadhiyar, Numerical libraries and the grid: The grads experiments with scalapack, Journal of High Performance Applications and Supercomputing 15 (4) (2001) 359–374.

[29] J. Schopf, Structural prediction models for high performance distriuted applications, in: Proceedings of Cluster Computing Conference (CCC'97), Atlanta, USA, 1997.

[30] J. Schopf, F. Berman, Performance prediction in production environments, in: Proceedings of 12th International Parallel Processing Symposium, Orlando, USA, 1998.

[31] J. Schopf, F. Berman, Using stochastic information to predict application behavior on contended resources, International Journal on Foundation in Computer Science 12 (3) (2001) 341–364.

[32] T. Sherwood, S. Sair, B. Calder, Phase tracking and prediction, in: ISCA '03: Proceedings of the 30th Annual International Symposium on Computer architecture, 2003.

[33] V. Taylor, X. Wu, J. Geisler, X. Li, Z. Lan, M. Hereld, I. Judson, R. Stevens, Prophesy: Automating the modeling process, in: Proceedings of the Third Annual International Workshop on Active Middleware Services, Tokyo, Japan, 2001.

[34] V. Taylor, X. Wu, R. Stevens, Prophesy: An infrastructure for performance analysis and modeling of parallel and grid applications, ACM SIGMETRICS Performance Evaluation Review 30 (4) (2003) 13–18.

[35] TeraGrid, http://www.teragrid.org.

[36] UK e-Science, http://www.rcuk.ac.uk/escience/default.htm.

[37] R. Wolski, Dynamically forecasting network performance using the network weather service, Journal of Cluster Computing 1 (1) (1998) 119–132.

[38] R. Wolski, N. Spring, J. Hayes, The network weather service: A distributed resource performance forecasting service for metacomputing, Journal of Future Generation Computing Systems 15 (5-6) (1999) 757–768.

[39] Z. Xu, X. Zhang, L. Sun, Semi-empirical multiprocessor performance predictions, Journal of Parallel and Distributed Computing 39 (1) (1996) 14–28.

[40] J. Yagnik, H.A. Sanjay, S. Vadhiyar, Performance modeling based on multidimensional surface learning for performance predictions of parallel applications in non-dedicated environments, in: ICPP '06: Proceedings of the 2006 International Conference on Parallel Processing, 2006.

[41] Y. Yan, X. Zhang, Y. Song, An effective and practical performance prediction model for parallel computing on nondedicated heterogeneous NOW, Journal of Parallel and Distributed Computing 38 (1) (1996) 63–80.

H.A. Sanjay received the BE degree in electrical and electronics engineering from the University of Kuvempu, India, in 1996 and the MTech degree in computer science and engineering from Visvesvaraya Technological University, India, in 2001. Currently he is pursuing a PhD at the Supercomputer Education and Research Center at the Indian Institute of Science, Bangalore. His research interests include grid computing, parallel and distributed systems, and performance modeling of parallel applications.

Sathish Vadhiyar is an assistant professor at the Supercomputer Education and Research Centre, Indian Institute of Science. He obtained his B.E. degree in the Department of Computer Science and Engineering at Thiagarajar College of Engineering, India, in 1997 and received his Master's degree in computer science at Clemson University, USA, in 1999. He obtained a PhD in the Computer Science Department at University of Tennessee, USA, in 2003. His research areas are in parallel and grid computing with primary focus on performance modeling of parallel applications, scheduling and rescheduling methodologies for grid systems, and grid applications. Dr. Vadhiyar is a member of IEEE and has published papers in peer-reviewed journals and conference proceedings. He was a tutorial chair and session chair of escience 2007 and served on the program committees of conferences related to parallel and grid computing including IPDPS, CCGrid, and eScience.