Master/Slave Computing on the Grid

Gary Shao *
Department of Computer Science
and Engineering
University of California, San Diego
San Diego, CA 92093-0114
gshao@cs.ucsd.edu

Francine Berman †
Department of Computer Science
and Engineering
University of California, San Diego
San Diego, CA 92093-0114
berman@cs.ucsd.edu

Rich Wolski †
Department of Computer Science
107 Ayres Hall
University of Tennessee
Knoxville, TN 37996-1301
rich@cs.utk.edu

Abstract

Resource selection is fundamental to the performance of master/slave applications. In this paper, we address the problem of promoting performance for distributed master/slave applications targeted to distributed, heterogeneous "Grid" resources. We present a work-rate-based model of master/slave application performance which utilizes both system and application characteristics to select potentially performance-efficient hosts for both the master and slave processes. Using a Grid allocation strategy based on this performance model, we demonstrate a performance improvement over other selection options for a representative set of Master/Slave applications in both simulated and actual Grid environments.

1. Introduction

The **master/slave** paradigm is a fundamental and commonly used approach for parallel and distributed applications. In master/slave applications, a single *master* process controls the distribution of work to a set of identically operating *slave* processes. The master/slave paradigm has been used successfully for a wide class of parallel applications [12][6][14], and is well suited as a programming model for applications targeted to distributed, heterogeneous "Grid" resources[1].

Methods which can improve the performance of master/slave applications are of considerable interest to many people. Researchers and application developers have previously experimented with tuning the granularity of master and slave processes to balance computation and communication, varying parameters such as the number and complexity of tasks assigned to slaves, and varying the number of slave processes used [3] [8][16]. Note that in a homogeneous environment, any processor can reasonably be chosen as a master or a slave, as all resources are typically considered to be equivalent. However, in a heterogeneous Grid environment, non-uniformity in both the peak and deliverable capacities of computational and communication resources can produce very different application execution times depending on which processor is chosen for the master and which processors are chosen for the slaves.

In this paper, we address the problem of how to determine a performance-efficient placement of master and slave processes running in shared, distributed and heterogeneous environments. In a heterogeneous environment, the choice of processor for the master can have a significant effect on total available work rate, directly impacting application performance. Our strategy for selecting a location for the master process involves identifying the host processor which allows for the largest aggregated system work rate, which we will define in the next section. Our strategy for selecting slaves utilizes the performance capacity of the available computation and communication resources to determine a

*Supported in part by NSF grant #ASC-9701333, DARPA/ITO contract #N66001-97-C-8531, NPACI award #ASC9619020

†Supported in part by NSF grant #ASC-9701333, DARPA/ITO contract #N66001-97-C-8531, NPACI award #ASC9619020

performance-ef£cient collection of workers.

This paper is organized as follows: Section 2 provides a performance model for distributed master/slave applications. Section 3 describes how we obtain and use input parameters for calculating resource work capacity values in our performance models. Section 4 describes our algorithm for selecting the resources to use for the master and slave processes. Section 5 gives a representative set of performance results from our experiments, Section 6 includes a short discussion of some related work, and Section 7 provides a summary of our work.

2. A master/slave performance model

We consider a model of master/slave applications in which the primary function of the master process m is to pass out and collect work from a set of slave processes $s \in S$. [1] We assume that communication patterns are simple and well-de£ned, requiring communication only between the master process and individual slave processes. We will de£ne the application's *work* as a divisible set of *tasks*; where each task may require some input data and produces some output data.

Tasks are completed in an application by progressing through four stages in the master/slave computation:

Stage 1 is the transmission of a command to initiate a task on one of the slave processes, including any data needed by the slave to perform the computation.

Stage 2 is the execution of the task by the designated slave.

Stage 3 is the transmission of results from the slave back to the master.

Stage 4 is any immediate processing of task results from the slave that must be done by the master.

While passing through each stage in the computation, a particular system resource must be employed by a task for some period of time, after which the task can move on to the the next stage. As an example, we can consider the simple network topology shown in Figure 1. If processor A in Figure 1 is designated as the master process, a task intended for slave processor B during Stage 1 will employ the use of network Net1 to transfer required data from processor A to processor B. During Stage 2 the task will utilize processor time on B to run task computations. During Stage 3 the task will again utilize network Net1 to transfer result data from B to A. Finally, during Stage 4 the task will utilize processor time on A to process the incoming results and to prepare for initiating additional task transfers to B.

[1] It would be straightforward to extend this work to the case in which the master may also perform some work as a slave.

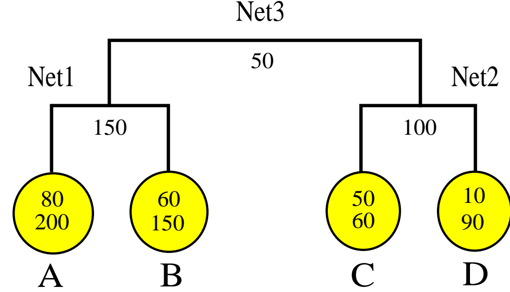

Figure 1. Example network con£guration.

In constructing a performance model for master/slave applications, we look at the rate that applications process tasks. *The rate at which an application cycles through tasks can be used as a measure of application performance, as faster overall cycle rates will correspond directly to reduced application execution times.*

If we consider the ¤ow of tasks between a master process m and a slave process s, we can make the de£nition:

$SlaveRate(m, s)$ is the task completion rate occurring between master m and slave s, in units of tasks per unit of time.

For master/slave computations where there is no communication between different slave processes, the total rate of task completions for an application will be the sum of the rates arising from task completions by individual slaves. We de£ne $AppRate(m, S)$ in Equation (1) to be the rate of task completions by an application with master process m and a set of slave processes S.

$$AppRate(m, S) = \sum_{s \in S} SlaveRate(m, s) \qquad (1)$$

We can then de£ne execution time, $ExecTime(m, S)$, for an application with a master process m and the set of slave processes S, and where $Tasks$ is the total number of tasks in the application.

$$ExecTime(m, S) = Tasks/AppRate(m, S), \qquad (2)$$

Application performance can thus be derived from values for $SlaveRate(m, s)$. One way to solve for these $SlaveRate$ values is to consider the system resource constraints which bound achievable application performance. To illustrate the concept, we go back to our simple example system in Figure 1, and observe that each processor and network has been labeled with one or more numerical values. We de£ne the numbers in the diagram to represent resource work capacities in terms of tasks per unit of time. The values next to network links represent network work capacity for that network link, the upper number within each circle

represents slave work capacity for that processor, and the lower number within each circle represents master work capacity for that processor.

Consider an application which uses processor C to host the master process. To solve for application performance, we would like to determine values for $SlaveRate(C, A)$, $SlaveRate(C, B)$ and $SlaveRate(C, D)$. The fundamental constraint condition to meet is that total task ¤ow rates through any resource cannot exceed the capacity value of that resource. This means, since task ¤ow from both processor A and processor B passes through network Net3 in our example, that the sum of $SlaveRate(C, A)$ and $SlaveRate(C, B)$ can be at most 50, the capacity of Net3.

In general, we can de£ne the following resource work capacity terms for processor and network resources. All terms are rates with units of tasks per unit of time.

$W_{MasterCPU}(i)$ = the maximum master work rate supported by a processor i. This is determined by processor i's capacity to perform Stage 4 computations for a speci£ed application.

$W_{SlaveCPU}(i)$ = the maximum slave work rate supported by a processor i. This is determined by processor i's capacity to perform Stage 2 computations for a speci£ed application.

$W_{Net}(n)$ = the maximum communication rate supported by a network n. This is determined by network n's capacity to perform the Stage 1 and Stage 3 communication of a speci£ed application.

Assuming we have a graph G representing network connectivity (such as the diagram in Figure 1) that allows us to identify which network resources are shared between different task ¤ows, and resource work capacity rates for each of the resources in our system, we can form a set of upper bounds on possible $SlaveRate(m, s)$ values. The process by which the network connectivity graph G and the work capacity rate terms can be derived for resources in a Grid environment will be discussed later in section 3.

First, to aid us in de£ning our upper bound constraints, we de£ne a helper set constructor function:

$ShareNet(G, S, m, n)$ takes as input a network connectivity graph G, a set of slaves processes S, a master process m, and a network resource n, and returns the set of slave processes from S which share the use of network resource n when communicating with m.

For master/slave applications, $ShareNet(G, S, m, n)$ can be easily determined for a network graph G, master process m, set of slaves S, and network resource n by following the single path in the graph G from each slave process $s \in S$ to the master process m, recording each path passing through the resource n.

Now we can give bounds which form constraints on application performance, as shown below[2].

$$\sum_{i \in S} SlaveRate(m, i) \leq W_{MasterCPU}(m) \quad (3)$$

$$SlaveRate(m, i) \leq W_{SlaveCPU}(i) \quad (4)$$

$$\sum_{i \in ShareNet(G,S,m,n)} SlaveRate(m, i) \leq W_{Net}(n) \quad (5)$$

Our goal is to £nd the values of $SlaveRate(m, s)$ which meet the constraints given above and which yield the largest value of $AppRate(m, S)$. The solution will correspond to a con£guration which delivers the best achievable application performance.

We can frame the problem of determining values for $SlaveRate(m, s)$ which yield the largest $AppRate(m, S)$ value as a **¤ow-rate problem** where: (1) the $SlaveRate(m, s)$ values are the ¤ows we wish to solve for, (2) m is the sink for all ¤ows, (3) the set S of slave processes are the sources for ¤ows, and (4) the ¤ow constraints correspond to the $W_{MasterCPU}(i)$, $W_{SlaveCPU}(i)$, and $W_{Net}(n)$ work capacities in our target environment.

Because the work ¤ows in a master/slave computation form a tree rooted at the master, and because we have limited our investigation to considering no more than one process hosted on each processor, ef£cient algorithms like the *Maximum-Flow algorithm* [5] exist for solving this problem. This approach can be used to solve the ¤ow-rate problem for several candidate processes m, £nding the one which is expected to deliver the maximum work ¤ow, and hence the best expected application performance. Section 4 describes the implementation of one such maximum-¤ow algorithm that can be used to £nd the largest possible work ¤ow.

3. Modeling work capacity rates in a Grid environment

In order to apply our work ¤ow performance model to real applications running in a Grid environment, we must derive a network connectivity graph G and appropriate values for the work capacity rate terms $W_{MasterCPU}(i)$, $W_{SlaveCPU}(i)$, and $W_{Net}(n)$.

The ¤ow-rate algorithm for determining application performance requires a graph G which represents the network connectivity between processor resources. For wide-area

[2] Since we consider here only cases where processors can host at most one process from the same application, we allow the process identi£er to be the identi£er of the processor hosting it in our inequality expressions.

Input Data	Description	Used In	How Acquired	When Acquired
$Graph_{Net}$	Network connectivity	G	ENV	periodically
$T_{SlaveCPU}$	CPU slave task time	$W_{SlaveCPU}$	benchmark	install
$T_{MasterCPU}$	CPU master task time	$W_{MasterCPU}$	benchmark	install
$Avail_{CPU}$	CPU availability	$W_{SlaveCPU}$, $W_{MasterCPU}$	NWS	run-time
$Size_{TaskXfer}$	Task data transfer size	W_{Net}	analysis, logging	application
BW_{Net}	Network bandwidth	W_{Net}	NWS	run-time

Table 1. Inputs for constructing the performance model.

Grid environments, it might be very dif£cult to get complete physical network con£guration data about every platform in the system. It is reasonable, however, to represent the target computational resources and their interconnection by a *logical view* which captures those areas where network constraints present potential bottlenecks to application performance. We derive a logical view of resource interconnection using a logical network con£guration discovery tool called Effective Network Views (ENV) [13]. (Other systems for discovery of effective system topology such as [9] might also be used.) The output of the ENV tool is a network graph representation where every processor belongs to a cluster of one or more machines. Machines in a cluster are connected together through a local network, where the capacity of the local network represents the limiting capacity of a network resource shared by each machine in the cluster. Clusters of local networks are connected together in our logical representation through a single layer of nonlocal network links. This representation is suitable for use in graph-based analysis techniques like our maximum ¤ow-rate problem, and directly translates to the network graph G in our ¤ow-rate solution.

The processor work capacity rates $W_{SlaveCPU}(i)$ and $W_{MasterCPU}(i)$ in our model are determined with two components: an application-speci£c component representing the maximum performance delivered by a processor resource in its unloaded state, and a dynamic component that is determined at run-time to adjust capacity rates to account for current loading conditions. The application-speci£c component is obtained by running a benchmark of the target application code on an unloaded processor, and measuring the times $T_{SlaveCPU}(i)$ and $T_{MasterCPU}(i)$ that are required to compute a single task on processor type i by the slave and master processes respectively. If the task computation time is variable over time, perhaps because of data dependencies in the application, we take an average value for all task times in one run of the application benchmark. This value could be scaled for particular classes of data sets at run-time if the variation in average task run times is large when different data sets are used. The benchmark times only have to be measured once for each platform type on which the application is built to run, so obtaining these values is computationally ef£cient.

The dynamic component of the work capacity terms for processor resources is calculated with the help of real-time monitoring and forecasting services such as the Network Weather Service [19] (NWS). The NWS provides real-time predictions of dynamic processor availability $Avail_{CPU}(i)$ (the percentage of CPU time a process can expect to get on processor i). $Avail_{CPU}(i)$ describes the predicted availability status of a processor resource, and can be generated independently from any particular application. This enables a single NWS system to provide simultaneous service to many applications requiring real-time information about resource behavior.

The processor work capacity rates can be calculated using the application-speci£c and dynamic components as shown below. The input parameters for these functions are summarized in Table 1.

$$W_{SlaveCPU}(i) = Avail_{CPU}(i)/T_{SlaveCPU}(i) \quad (6)$$

$$W_{MasterCPU}(i) = Avail_{CPU}(i)/T_{MasterCPU}(i) \quad (7)$$

The network work capacity rate $W_{Net}(n)$ in our model is also calculated using two components. One component is the application-speci£c term $Size_{TaskXfer}$, which represents the amount of data transferred between a master process and a slave process for each task in an application.

If the task data transfer sizes are a variable quantity over time, perhaps due to data dependencies in the application, we must calculate an average data transfer value that represents expected steady-state communication behavior over the time of an entire application run.

The second component used in calculating network work capacities on a network resource n is a dynamic prediction of expected available network bandwidth $BW_{Net}(n)$, which we obtain from the NWS. The network work capacity rates can be calculated using application-speci£c and dynamic components as shown below. The input parameters are again summarized in Table 1.

$$W_{Net}(n) = BW_{Net}(n)/Size_{TaskXfer} \tag{8}$$

Having constructed a set of resource constraint values to help model the performance of a Grid environment, we should also discuss an obvious limitation of our approach. For each of the terms derived in this section, we have generated an average-value expression for use in our steady-state application performance model. Yet each of the properties being modeled might in reality exhibit considerable variability over time, either due to time-varying load conditions or data dependent behavior of the application being run. Our experience has been that despite the limitations of converting many variable terms to average steady-state values, our approach still yields a performance model which can do a good job at estimating application performance, and which provides an effective tool for helping to solve the resource selection problem, which we discuss next.

4. Selecting a master and the slaves

Given the work-rate performance model described in Section 2 and a logical representation of the work capacities of Grid resources, we can now consider strategies for selecting processors to host the master and slave processes. These are important issues for master/slave applications running in Grid environments, where users may be able to choose from among many different types of resources, and where availability of these resources may change over time.

Selection of the right processor to host the master process can signi£cantly impact the overall application performance, as the following section will show. Knowing which master placement produces the best application performance might also in¤uence other important decisions, such as where to ef£ciently position input and output £les for the application. Selection of the right set of processor resources to host the slave processes has two goals: (1) selecting enough resources from the available set to produce the best achievable application performance,and (2) limiting the selection to resources that will actually bene£t application performance. The second goal is important for Grid environments where resources can be shared by many users, and resources can be owned and managed by many different organizations. In these environments, it is desirable that applications use only those resources they really need; thereby allowing limited pools of shared resources to satisfy the largest number of users. We will £rst consider the issue of selecting the right host for the master process.

4.1. Master selection example

In a heterogeneous system, selection of a location for the master process very strongly depends on the deliverable work capacity of candidate resources. Consider the logical Grid con£guration shown back in Figure 1, where four processors are connected by a system of three networks. We have labeled the network resources with values representing the W_{Net} capacity terms. The processor resources, shown as circles in the diagram, have been labeled with two values: a $W_{SlaveCPU}$ capacity term on top, and a $W_{MasterCPU}$ capacity term on the bottom. All capacity terms are in units of tasks per second.

For this simple example system, we can determine the assignment of the master process to a processor that gives us the greatest achievable work ¤ow. If processor A is selected to host the master process, processor B is able to provide 60 tasks/sec work rate as a slave. In addition, a maximum of 50 tasks/sec worth of data can be transferred over network Net3, a work rate which can be supplied by processor C. The total expected application work rate with processor A hosting the master is therefore 110 tasks/sec. If we consider selecting processor C to host the master process, we observe that processor D can deliver a work rate of 10 tasks/sec working as a slave. In addition, we can transfer a maximum of 50 tasks/sec worth of data over network Net3, which can be supplied by processor A. It becomes apparent that processor C is constrained from achieving any higher application work rate by the limitations on the Net3 capacity, as well as the capacity of processor C to serve as the master host, to no more than 60 tasks/sec. We could proceed in a similar manner for all the processors, and derive expected application work rates for each candidate. Table 2 shows one set of possible outcomes for this process. It is apparent from the last column in Table 2 that processor B is the best choice, yielding a potential application work rate of 130 tasks/sec.

4.2. Selecting the master

More generally, we have developed a basic algorithm for £nding the best performing host for the master process. It is based on the well-known maximum-¤ow algorithm by Ford and Fulkerson [7]. In this algorithm, we keep augmenting the estimated ¤ow rate for each master host by adding the

Master Location m	$W_{MasterCPU}$ (m)	$SlaveRate$ (m, A)	$SlaveRate$ (m, B)	$SlaveRate$ (m, C)	$SlaveRate$ (m, D)	$AppRate(m)$
A	200	0	60	50	0	110
B	150	80	0	50	0	130
C	60	50	0	0	10	60
D	90	40	0	50	0	90

Table 2. Work rates resulting from master placement decision.

contributions of slave processors. Additional contributing slaves are selected £rst from those on the same local network as the master. This continues until either all of the slaves have been included, or no further slave work rates can be incorporated because of either capacity limitations on network resources, or capacity limitations of the master processor itself. If further capacity is available from processors on non-local networks, they are added one by one to the accumulated master total until no further additions are possible without exceeding one of the resource capacities. Figure 2 illustrates our basic algorithm for £nding the best performing master host. Upon termination of the algorithm, the processor with the highest calculated work rate is selected as the master.

4.3. Complexity

In deriving the complexity of our algorithm, we note that our simpli£ed logical representation of network con£guration reduces the entire system to sets of processors connected by local networks. Each of these local networks is then connected to other local networks by at most one level of remote networking. With this logical topology, data transfers between slaves on the same local network pass through only one level of networking, and encounter only one network resource constraint. Data transfers between slaves located on different local networks will pass through at most three levels of networking, and must satisfy at most three networking constraints. All slave work rates must meet the resource constraints of the master processor. With this arrangement, there are at most four tests of constraints in our algorithm that have to be checked for each master and slave pairing.

If we have n processors in our system, then each master candidate can have at most $n-1$ slaves, and each individual master work rate calculation takes $O(n)$ time to calculate. Calculating maximum work rates for all n possible master candidates thus takes $O(n^2)$ time. Since our algorithm requires only simple compare and accumulation operations for each resource constraint test, the entire algorithm is ef£cient for the numbers of processors and networks we currently £nd in Grid environments available to a typical user.

4.4. Selecting the slaves

After selecting the master processor, we turn to selection of the slave processors. The issue is to select a set of processors for hosting slave processes that will deliver good aggregate performance. One approach is to start with the set of slave processors found in our master selection algorithm that yielded the highest expected application performance. Our algorithm keeps track of this set in the $Found(m)$ list, a list containing slaves used by the algorithm to calculate the maximum work rate for an application with processor m as the master host. Our master selection algorithm ensures that this set of processors results in work ¤ows that fall within the constraints imposed by resource capacity limitations.

In numerous experimental trials using the set of processors from $Found(m)$ as slave hosts, we observed that the slave processors were often not delivering the maximum work rate values we expected in our algorithm. Observations of selected slaves showed the reduction in slave performance was due to the presence of unaccounted idle time, periods of time when slave processors were not doing useful work. An explanation for the observed idle times comes from observing the manner in which tasks are distributed to slave processors from the master. Each master/slave application we tested maintained a queue of available tasks on the master process, and distributed new tasks to individual slave processes upon request (a very commonly used technique). Because of contention for shared resources, such as networks and the master processor, delays sometimes occurred between the time a slave processor £nished one task and the time at which the next task appeared for processing. These delays appeared as idle time in our observations of the slaves. With a minimum set of slaves selected to achieve the desired work rate, the unexpected idle time in the slaves resulted in a reduction of the actual total work rate achieved.

The work ¤ow-rate performance model correctly determines possible application performance based on resource capacity limits. Our master selection algorithm uses this performance model, and in the process identi£es a set of slaves which delivers this performance, assuming that each slave delivers its maximum work rate. Experimentation has shown that sometimes these slaves actually deliver less than their predicted maximum work rates, resulting in less per-

For all networks k
    Calculate maximum network capacity $W_{Net}(k)$
For all processors j
    Calculate maximum master processor capacity $W_{MasterCPU}(j)$
    Calculate maximum slave processor capacity $W_{SlaveCPU}(j)$
For each candidate master processor p on local network n
    Set sum for candidate slave work rates $CandRate(p) = 0$
    Set found set $Found(p)$ to empty
    For all networks k
        Set network utilization sum $NetUtil(k) = 0$
    Get maximum capacity $W_{Net}(n)$ of local network n
    Get maximum master processor capacity $W_{MasterCPU}(p)$
    While $CandRate(p) < W_{Net}(n)$ and $CandRate(p) < W_{MasterCPU}(p)$
        Select new processor s from same local network as p with
        the largest available $W_{SlaveCPU}(s)$ value
        Get slave processor capacity $W_{SlaveCPU}(s)$
        Get fraction F of $W_{SlaveCPU}(s)$ that will not cause
        utilization $NetUtil(n)$ to exceed $W_{Net}(n)$
            Add F to $CandRate(p)$
            Add F to $NetUtil(n)$
            Add processor s to found set $Found(p)$
    Total candidate work rate $CandRate(p) = \min(CandRate(p), W_{MasterCPU}(p))$
    Total local network utilization $NetUtil(n) = CandRate(p)$
    While $CandRate(p) < W_{Net}(n)$ and $CandRate(p) < W_{MasterCPU}(p)$
        Select new processor q from outside local network with
        the largest available $W_{SlaveCPU}(q)$ value
        Get slave processor capacity $W_{SlaveCPU}(q)$
        Get fraction F of $W_{SlaveCPU}(q)$ that will not cause
        utilization $NetUtil(i)$ to exceed $W_{Net}(i)$ for any network i
            Add F to $CandRate(p)$
            Add F to $NetUtil(n)$
            Add F to other $NetUtil(k)$ where network k is involved in
            communications between processors p and q
            Add processor q to found set $Found(p)$
Select processor p with largest $CandRate(p)$ as master

Figure 2. Algorithm for £nding best processor for the master.

formance than resource capacity constraints would allow. One way to get application performance back up to predicted levels is to add additional slave processors to the originally selected mix, thereby raising the effective slave work rates achieved up to expected values. Our goal is to compensate for lost performance due to idle time on the individual slave processors, while keeping the number of additional processors down to the minimum needed to accomplish this goal.

Our steady-state ¤ow-rate performance model was not useful in helping to decide how many slaves to add to increase effective performance because it could not account for idle times caused by slaves waiting for new tasks to arrive. To address this shortcoming and others in our steady-state approaches to performance analysis, we developed a master/slave application performance simulator to provide signi£cant new capabilities. We discuss this simulator and how it can be used to help solve the slave selection problem in the following subsections.

4.5. An application performance simulator

We originally developed a master/slave application performance simulator to provide detailed predictions of performance and resource behavior for applications running in Grid environments. One effective use we have found for this simulator is to help determine how many additional slave processors might be added to a predicted group of master and slave processors to make up for performance losses due to slave idle time.

At its core, our simulator is a set of routines which model the behavior of tasks as they pass through a system comprised of two kinds of resources: processors and networks. The resources are modeled as single servers with £rst-in-£rst-out input queues. Service times for the processor resources determine how long a task has control of the processor before relinquishing the resource to the next task in the input queue, and are dependent on the same processor availability parameters $Avail_{CPU}(i)$ and estimated task execution times $T_{SlaveCPU}(i)$ and $T_{MasterCPU}(i)$ developed earlier for our ¤ow-rate model. Service times for the network resources determine how long a network resource is committed to servicing data transfers for each task, and are dependent on the same network bandwidth parameters $BW_{Net}(n)$ and size of the data transfers values $Size_{TaskXfer}$ developed for the ¤ow-rate model presented earlier. In addition, all of the parameters can be adjusted to use either static steady-state values like those in the ¤ow-rate performance model, or more dynamic data inputs such as statistical distributions or actual measured trace values from application runs. Network connectivity is represented using the same graph G, an output of the ENV tool, used in the work ¤ow-rate performance model.

The simulator is written in highly portable C-language code, with the help of a simulation library package called Sim++ [4]. This simulator can be easily embedded into other programs, such as an application scheduler, to provide detailed predictions of application performance and resource utilization levels. It is particularly useful for observing the performance impact of changing application or resource parameters.

4.6. Using simulation to enhance slave selection

Our algorithm for £nding the correct set of slave processors starts with the master processor m and the $Found(m)$ set of slaves from the master selection algorithm. The simulator is run with these machines as the target environment, using the same values for resource capacities as were used in the master selection algorithm. Results from the simulation are checked to see if any idle time on the simulated slaves results in a signi£cant decrease in overall application performance. If a substantial performance decrease is found, resource utilization £gures from the simulation are checked to see where additional processors might be added without exceeding existing resource constraints. If more slave processors are available to be added that will not violate any known resource constraints, they are added to the set of found slaves. A new system con£guration with the additional processors added in is constructed and simulated once again. The process of slave additions and testing by simulation repeats until either there are no further performance gains realized by adding more slave processors, or no more processors can be found and placed without exceeding one of the known resource capacity constraints. Figure 3 illustrates our algorithm for £nding the set of slave processors.

The algorithm given above makes good use of simulator results which calculate predicted resource utilization values for every resource in the system. These values allow us to quickly identify where in the system, if anywhere, slave processors might be added to improve application performance. In practice, the number of times the simulation cycle needs to be run is small as the process quickly converges to a situation where either additional performance gains are insigni£cant, or no further additions can be made without exceeding a resource constraint.

5. Experimental results

In this section, we describe experiments whose goal it is to test the usefulness and accuracy of our work-rate performance model and application performance simulator, as well as the performance of our algorithms for selecting master and slave processors.

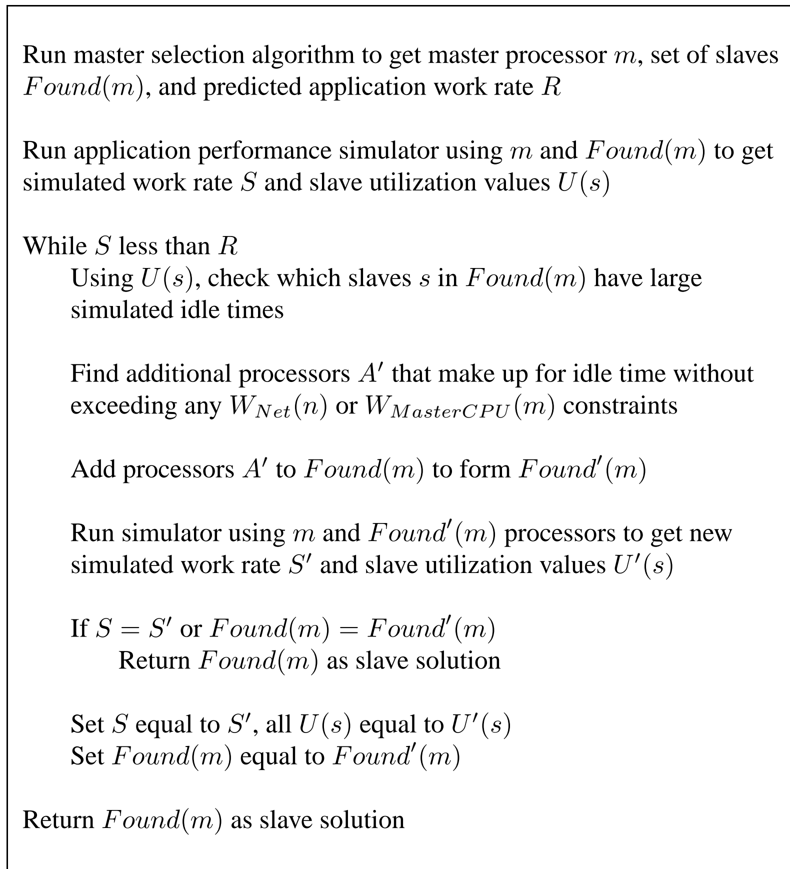

Figure 3. Algorithm for finding best processors for slaves.

We use as an application test suite three applications chosen to represent a spectrum of potential master/slave distributed applications. The applications were selected and implemented to test the sensitivity of our approach to computation and communication granularity. Our master/slave implementation of the Mandelbrot image application is expected to display a relatively high sensitivity to communication constraints, as the amount of image data transferred during execution is large compared to the overall computation time. At the other extreme is the NAS Parallel Benchmarks' EP [18] application, which performs relatively little data transfer compared to the time spent computing. The Povray [11] ray-tracing application falls somewhere in the middle, with the transfer of one fourth the amount of image data as the Mandelbrot application which was spread out over a longer computation time.. Each of the applications was initially benchmarked on all target processor types to produce the application-specific parameters needed for our performance analysis tools. The applications are summarized in Table 3.

5.1. Experimental design

In the experiments, we compared *predicted execution time* (resulting from our performance model), *simulated execution time* (using the application simulator), and *actual execution time* (determined from experimental runs). All comparisons were made in a non-dedicated environment where the load traces used for the predicted and simulated execution times were determined from the NWS load trace of the actual execution time runs. We used identical parameter inputs for network configuration, resource constraints, and application characteristics in both work-flow analysis and performance simulation tools. In this way, we attempted to compare each set of execution times under the same environmental conditions.

The target experimental platform was a heterogeneous mix of Intel processor-based machines running Linux, and Sun SPARC machines running Solaris located in the Parallel Computation Laboratory in the Department of Computer Science and Engineering at the University of California, San Diego. The experiments were run with all machines in non-dedicated mode, but outside loading from compet-

Name	Description	Emphasis
Mandelbrot	parallel fractal image generator	communication
Povray	parallel implementation of popular ray-tracer	both
NBP EP	NAS Parallel Benchmark EP variant	computation

Table 3. List of applications used in experiments.

ing jobs was observed to be relatively light for most of the machines during the course of experimentation.

5.2. Results

In the £rst set of experiments, we ran the test suite of applications on a set of nine workstations shown in Table 4. For the three applications, trials were run with each of the nine processors being selected to run as the master while the other eight were included to run as slaves. In all cases, the work ¤ow-rate problem was solved for each con£guration of master and slaves to give the expected application execution time, shown as the light bars in Figures 4-6. The application performance simulator was run for all cases to give a predicted application execution time, shown by the middle bars in the graphs. And £nally, the real applications were run on each con£guration and their execution times recorded to appear as the dark bars on the graphs.. Figure 4 shows the results while running the relatively communication-heavy Mandelbrot application. Figure 5 shows the same set of execution times for the more balanced Povray application, while Figure 6 shows execution times for the computation-intensive NAS Parallel Benchmarks' EP application.

In these experiments, the work-rate performance model would have done a good job of identifying the correct master host to produce the fastest application execution times. In the Mandelbrot series of experiments, the machine *thing1* was calculated to yield the lowest execution time, which was con£rmed in the actual application run. For this application the highest execution time, achieved with the machine named *lorax*, took 170% longer to £nish than the best choice. For the other two applications, the work-rate performance model estimates of execution time again showed results which correlated closely with actual application run times. For these applications, which exhibited lower dependence on network constraints, the differences between the worst and best performers was smaller: about 25% for Povray and 10% for NAS EP. The work-rate based performance model correctly ordered master performance for both communication and computation constrained applications. The results also show that the application performance simulator did a good job of tracking the actual application execution times as well.

The experimental results show a small number of cases where the execution time was signi£cantly underestimated for the Mandelbrot application. Analysis of experimental results leads us to believe the discrepancy in predicted and actual performance on the communication-heavy application was due to inadequate benchmarking of the $W_{MasterCPU}$ constraint terms. Actual application performance is worse than that predicted by both the work-¤ow model and the simulator because both tools overestimated the capacity of the single master process to process incoming data and respond to new task requests. When the real master process fails to keep up with projected work rates, the overall application work rate is reduced and execution time becomes relatively larger. Improved methods for benchmarking master processor performance are currently being developed to overcome this shortcoming.

In the second set of experiments, we look at two of our applications: Mandelbrot and Povray. In these trials we pick a speci£c host for the master process, then run our application for different numbers of slave processes. We show measured execution times and simulated execution times for our two applications as we increase the number of slave processors.

Figure 7 shows results with our communication-intensive Mandelbrot application for two different choices of the master host. These results show that the number of slaves which can bene£cially be employed varies under different conditions, and is heavily constrained by the network speed of the master process host. Figure 8 shows results with the Povray application, whose performance is less dominated by communication costs. In our test environment, this application shows more scalable performance than Mandelbrot, but eventually also reaches a point where additional processors do not signi£cantly decrease execution time. Results are shown for only one master case because data for other cases produces almost identical graphs. Results for our third application, NPB EP, are not shown here, but they are very similar to those for povray, with simulation predicted run times and actual application run times very close for all numbers of processors. These results indicate that for our representative examples, the performance simulator can be a useful tool to help predict the points at which either additional slaves should be added to a computation to increase performance, or when additional slaves cease to have any useful effect.

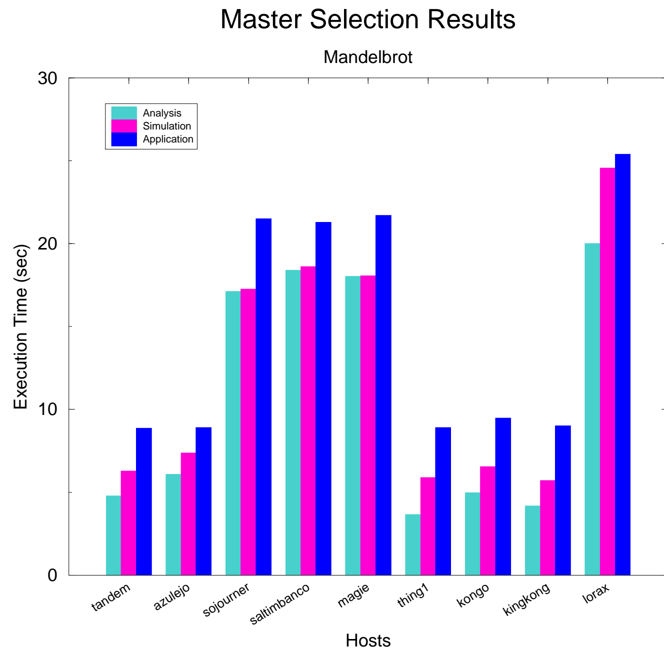

Figure 4. Execution time of communication-intensive application while varying master host.

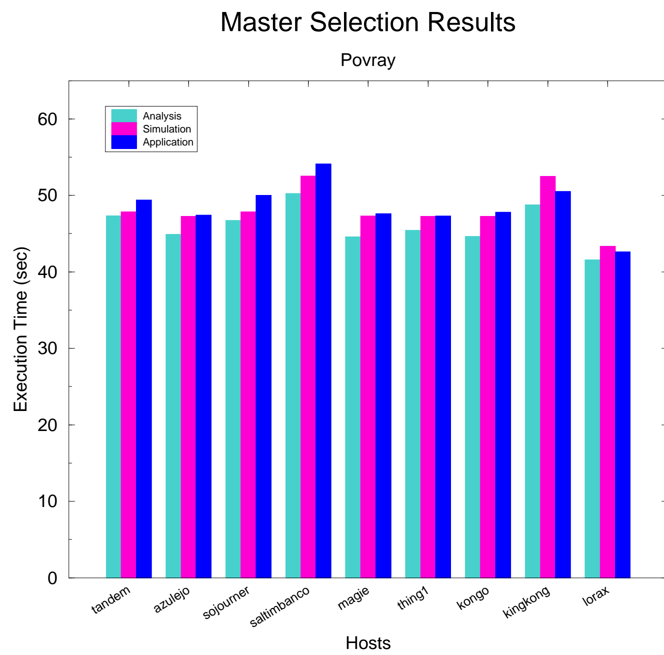

Figure 5. Execution time of application while varying master host.

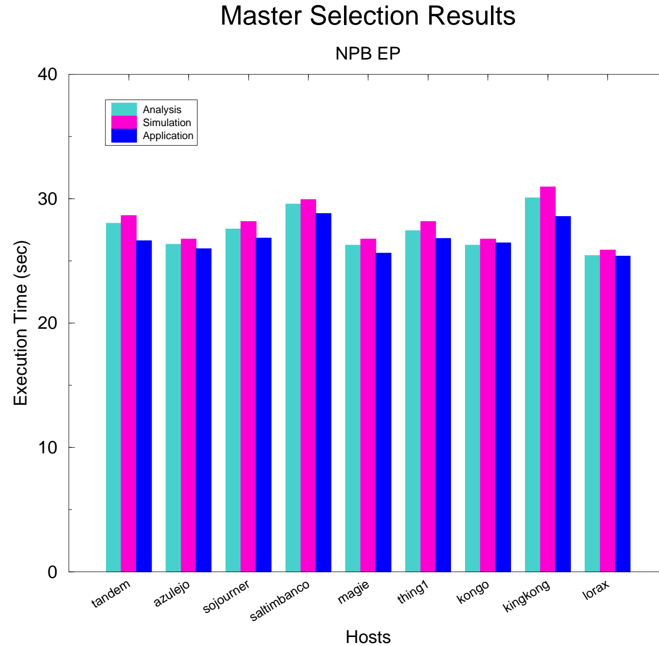

Figure 6. Execution time of computation-intensive application while varying master host.

Name	Processor	Network	OS
azulejo	Intel Pentium Pro 200	100 Mbit/s ethernet	Linux 2.0.36
kingkong	Sun UltraSPARC-IIi 333MHz	100 Mbit/s ethernet	Solaris 2.6
kongo	Sun UltraSPARC 166MHz	100 Mbit/s ethernet	Solaris 2.6
lorax	Sun microSPARC II 85MHz	100 Mbit/s ethernet	Solaris 2.6
magie	Intel Pentium Pro 200	10 Mbit/s ethernet	Linux 2.1.125
saltimbanco	Intel Pentium II-400	10 Mbit/s ethernet	Linux 2.1.125
sojourner	Intel Pentium II-266	10 Mbit/s ethernet	Linux 2.2.9
tandem	Intel Pentium II-300	100 Mbit/s ethernet	Linux 2.0.36
thing1	Sun UltraSPARC 200MHz	100 Mbit/s ethernet	Solaris 2.6

Table 4. Partial list of heterogeneous mix of machines used in experiments.

6. Related Work

Many different approaches to predicting the performance of parallel applications on distributed-memory machines have appeared in the literature. A partial summary of some earlier efforts can be found in [10]. Unfortunately, these approaches often suffered from either limited accuracy under real-world conditions (caused by making many simplifying assumptions), or from excessive complexity when either constructing or using the models. Our approach to performance prediction focuses on achieving useful levels of prediction accuracy while limiting model complexity and allowing ef£cient measurement and quanti£cation of important model parameters.

The application of performance prediction to the problem of resource selection has also been addressed recently by Weissman and Zhao [17]. In their work, Weissman and Zhao use heuristics to select a number of candidate con£gurations, then employ cost functions to derive computation and communication times for each con£guration. They then select the con£guration yielding the lowest total cost. Our approach to resource selection ef£ciently evaluates application performance for different con£gurations using only simple constraint calculations.

Subhlok, Lieu and Lowekamp [15] have looked at automatically selecting processor nodes for applications running on high-speed networks. For their results, Subhlok, Lieu and Lowekamp present algorithms which allow them to automatically select nodes with three different goals: maximizing computation capacity, maximizing communi-

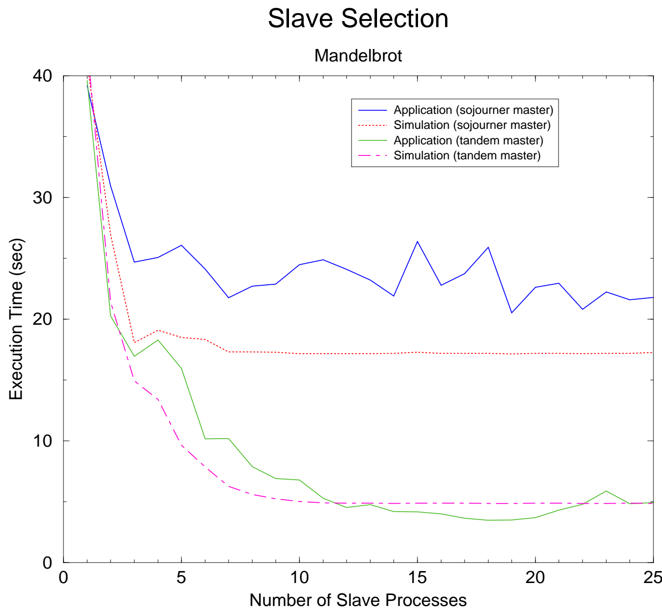

Figure 7. Application performance with varying numbers of slaves.

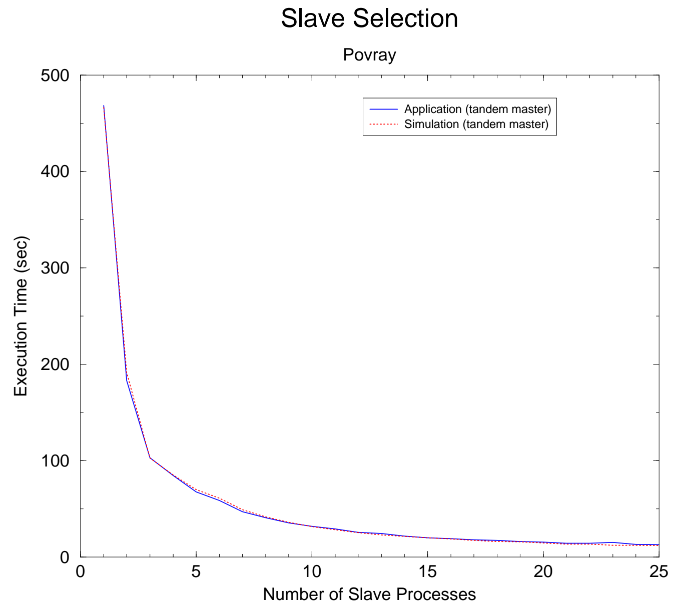

Figure 8. Application performance with varying numbers of slaves.

cation capacity, or balancing computation and communication. Their paper does not explain how the correct goal is selected to match speci£c application characteristics in order to give optimum performance. Our approach automatically determines performance bottlenecks based on both computation and communication constraints, and £nds the best performing con£guration for all cases.

7. Summary

In this paper, we have described a rate-based performance model for master/slave applications running on distributed heterogeneous processors and networks. By parameterizing this steady-state performance model with some dynamic run-time information, we are able to accurately predict maximum achievable application performance rates – even in the cases where application characteristics and resource behavior are not steady over time.

We have also described an application performance simulator which accurately simulates the dynamic interaction of a master/slave application with a de£ned con£guration of performance constrained resources. This simulator allows for a detailed analysis of where performance bottlenecks due to resource limitations may occur in an application. This kind of detailed information about how applications interact with resources in a Grid environment can be very valuable for resource selection at application runtime, advanced application and platform planning, and program development activities. The key to our success with our performance prediction tools has been the identi£cation of a common set of application and resource parameters which could be quanti£ed and measured, and which captured both the static and dynamic aspects of application performance in Grid environments.

Based on the effectiveness of our performance prediction tools, we have developed algorithms for master and slave resource selection on Grid platforms. These algorithms enable the selection of a master processor and a set of slave processors which allow maximum application performance to occur. Actually achieving the maximum application performance in dynamic Grid environments may also require the use of other run-time techniques to handle issues like load balancing and fault tolerance. These are issues we are actively researching, and will be the subject of future publications.

Some brief experimental data was presented to verify that both our performance prediction tools and our strategies for selecting master and slave resources were sound. We are currently integrating the performance tools and resource selection strategies into an AppLeS [2] Grid application scheduler with the goal of providing an automatic mechanism for high-quality distributed master/slave scheduling in heterogeneous and dynamic Grid environments.

In the future, we would like to extend the work-rate-based performance model to other common classes of paral-

lel computing in Grid environments. We would also like to study whether other physical resource characteristics, such as available memory, might be bene£cial to include in our constraint analyses. Our experience has shown that the idea of estimating application performance by accounting for application/resource constraints appears promising as a tool for enabling more effective application scheduling.

References

[1] F. Berman. High-performance schedulers. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, chapter 12. Morgan Kaufmann Publishers, July 1998.

[2] F. Berman and R. Wolski. Scheduling from the perspective of the application. In *Proceedings of the Fifth IEEE Symposium on High Performance Distributed Computing*, pages 100–111, Aug. 1996.

[3] A. Clematis and A. Corana. Performance analysis of task-based algorithms on heterogeneous systems with message passing. In *Proceedings Recent Advances in Parallel Virtual Machine and Message Passing Interface, 5th European PVM/MPI Users' Group Meeting*, Sept. 1998.

[4] R. M. Cubert and P. Fishwick. *Sim++, Version 1.0*. Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, 1995.

[5] J. R. Evans and E. Minieka. *Optimization Algorithms for Networks and Graphs*, chapter 5, pages 178–233. Marcel Dekker, Inc., second edition, 1992.

[6] K. Everaars and B. Koren. Using coordination to parallelize sparse-grid methods for 3-d cfd problems. *Parallel Computing*, 24(7):1081–1106, 1998.

[7] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, New Jersey, 1962.

[8] D. Gelernter, M. R. Jourdenais, and D. Kaminsky. Piranha scheduling: Strategies and their implementation. *International Journal of Parallel Programming*, 23(1):5–33, Feb. 1995.

[9] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok. A resource query interface for network-aware applications. In *Proceedings of Seventh International Symposium on High Performance Distributed Computing*, July 1998.

[10] W. Meira. Modeling performance of parallel programs. Technical Report 589, Computer Science Department, University of Rochester, Rochester, NY, June 1995.

[11] Persistence of vision raytracer. Persistence of Vision Development Team, 1999. http://www.povray.org/.

[12] J. Pruyne and M. Livny. Interfacing condor and PVM to harness the cycles of workstation clusters. *Future Generation Computer Systems*, 12(1):67–85, 1996.

[13] G. Shao, F. Berman, and R. Wolski. Using effective network views to promote distributed application performance. In *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications*, June 1999.

[14] L. M. Silva, V. Batista, P. Martins, and G. Soares. Using mobile agents for parallel processing. In *Proceedings of the International Symposium on Distributed Objects and Applications*, Sept. 1999.

[15] J. Subhlok, P. Lieu, and B. Lowekamp. Automatic node selection for high performance applications on networks. In *to appear in Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1999.

[16] A. S. Wagner, H. V. Sreekantaswamy, and S. T. Chanson. Performance models for the processor farm paradigm. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):475–489, May 1997.

[17] J. B. Weissman and X. Zhao. Scheduling parallel applications in distributed networks. *Journal of Cluster Computing*, 1(1), 1998.

[18] S. M. White, A. Alund, and V. S. Sunderam. Nas parallel benchmark kernels for pvm 3. http://www.nas.nasa.gov/NAS/NPB/, Oct. 1993.

[19] R. Wolski. Dynamically forecasting network performance using the network weather service. In *Proceedings of the 6th High-Performance Distributed Computing Conference*, pages 316–325, Aug. 1997.

Gary Shao is a graduate student in the Department of Computer Science and Engineering at the University of California, San Diego. His research interests include parallel and distributed computing, adaptive scheduling, and application development environments. He received his B.S from the University of Missouri, Columbia and his M.S. from Washington University in St. Louis, Missouri.

Francine Berman is a Professor of Computer Science and Engineering at the University of California, San Diego. She is also a Senior Fellow at the San Diego Supercomputer Center, Fellow of the ACM, and founder of the Parallel Computation Laboratory at UCSD. Her research interests over the last two decades have focused on parallel and distributed computation, and in particular the areas of programming environments, tools, and models that support high-performance computing. She received her B.A. from the University of California, Los Angeles, her M.S. and Ph.D. from the University of Washington.

Rich Wolski is an Assistant Professor in the Department of Computer Science at the University of Tennessee and a partner in the National Partnership for Advanced Computational Infrastructure. His research interests include parallel and distributed computing, on-line performance analysis techniques and software, compiler runtime system, and dynamic scheduling. He received his B.S. from the California Polytechnic University, San Luis Obispo and his M.S. and Ph.D. from the University of California at Davis/Livermore Campus.