Proc. 1996 Computer Measurement Group Conference (CMG), San Diego, CA, Dec. 8-13, 1996.

PERFORMANCE PREDICTION OF PARALLEL APPLICATIONS ON NETWORKS OF WORKSTATIONS

Daniel A. Menascé*
Dept. of Computer Science
George Mason University
Fairfax, VA 22030
menasce@cs.gmu.edu

Amar Rao
Science Applications International Corporation
1953 Gallows Rd,
Vienna, VA 22182
Amar.B.Rao@cpmx.saic.com

Abstract

Networks of workstations (NOWs) are an alternative to supercomputers when it comes to running parallel applications. In most environments, users of a workstation use less than 20% of the available CPU cycles. The remaining capacity can be used to support tasks of a parallel application. This paper presents a hybrid analytic and simulation model that can be used to predict the performance of parallel applications specified by a task graph on a network of workstations. The model was validated through measurements and takes into account the specific workstations used, the scheduling discipline, and the network characteristics. Moreover, it considers the effect of the owner's interactive workload on the execution time of the parallel application. The paper briefly discusses a tool developed by the authors to implement the model.

1 Introduction

The idea of exploiting a Network of Workstations (NOW) to execute processes other than the ones submitted by the owner of the workstation arises from studies which have shown that a large percentage of a workstation CPU cycles are idle for over 80% of the time on the average [8]. Although the idea of a network of workstations itself is not new, it is now possible because of recent technological advances in the areas of:

- *Networks* : new networking technologies, like switched local area networks, allow bandwidth to scale with the number of workstations and low overhead protocols have made fast communication possible.
- *Workstations* : present day workstations are very powerful machines. A top end workstation today operates at almost one third the speed of a supercomputer or a massively parallel machine and costs less. Also a typical workstation offers large inexpensive memory and disk capacity which can be accessed across the network, making the resources of a workstation useful and accessible to users other than the owner of the workstation.

There are several opportunities presented by network of workstations for different needs [1]. For NOWs to succeed as an alternative to supercomputers (and PCs), they should deliver at least the interactive performance of a dedicated workstation while providing the aggregate resources of the network for sequential and parallel workloads which demand these resources. This requires a resource allocation policy that explicitly preserves interactive performance, while allowing dedicated and unused resources throughout the network to be used by demanding applications. These resources include DRAMs for memory-intensive programs, disks for I/O bound programs, and CPU for parallel and distributed programs.

A careful resource allocation and scheduling policy is necessary to ensure that idle cycles are easily located and allocated to tasks that require them. Although the principle behind global scheduling in a network of workstations is the same as in any distributed system, the NOW environment poses additional constraints. In a typical workstation environment, jobs submitted by the owner of a workstation are typically interactive. The owner submits a command to the workstation and waits for the reply. After getting the reply, the owner spends some time (called the think-time) before issuing another command. As mentioned earlier, the commands issued by the owner of a workstation use less than 20% of the CPU time [2, 5]. On the whole however, the overall load distribution can vary greatly in magnitude over time and is not homogeneous in nature. Hosts may be idle or running interactive applications or

*This work was supported in part by the Advanced Research Project Agency (ARPA) under contract DABT63-93-C-0026.

compute-intensive batch jobs or both. Also, the set of machines which are available as hosts for remote execution is constantly changing. These conditions make it difficult to employ static scheduling policies for job-scheduling in a NOW. As mentioned before, the main requirement of a NOW to succeed as a "Total Solution System" is that it should deliver at least the interactive performance of a dedicated workstation while providing aggregate resources of the network for demanding applications. Any scheduler for a Network of Workstations has to account for all the above factors and perform processor allocation accordingly. Processor (re)allocation can be done by either preempting a running process, or by not doing so. In the case of preemptive transfers, the process that is currently running is stopped, transferred to another processor, where it is restarted. Condor [5, 6] and Stealth [2] are two examples of schedulers for distributed applications in a network of workstations.

The performance of a parallel application running on a network of workstations depends on the number and type of workstations used to support the application, on the owner workload at each workstation, on the network used to connect the workstations, and on the scheduling algorithm used to allocate parallel tasks to workstations. This paper presents a hybrid analytic and simulation model that can be used to predict the performance of parallel applications specified by a task graph on a network of workstations. The model takes into account the specific workstations used, the scheduling discipline, and the network characteristics. Moreover, it takes into account the effect of the owner's interactive workload on the execution time of the parallel application. The paper briefly discusses a tool developed by the authors to implement the model.

The paper is organized as follows. Section two presents some basic assumptions and definitions. Section three describes the performance model. Section four presents the results of the experiments conducted to validate the model. Section five presents some performance prediction studies carried out with the validated model. Section six presents a brief description of SimNOW—a predictive performance analyzer for NOWs designed and implemented by the authors. Finally, section seven presents some concluding remarks.

2 Basic Assumptions

The environment considered here consists of W workstations connected through a local network. Each workstation has an owner that generates an interactive workload. Tasks belonging to a parallel application may be running at one or more workstations. It is assumed that the processes generated by the owner of the workstation have preemptive priority over the parallel tasks running on it.

The interactive workload at workstation i is characterized as a closed workload composed of N_i processes that alternate between processing and think time periods. The average duration of the think time at workstation i is Z_i and the average CPU service demand of an owner-generated process at workstation i is D_i. I/O is not considered in this study but could easily be included with straightforward extensions. Our model allows for heterogeneous workstations. A parallel program P_i is characterized by the tuple $(\mathcal{T}, \mathcal{R}, \mathcal{D}, \mathcal{C})$ where

- $\mathcal{T} = \{T_1, \cdots, T_n\}$ is a set of n tasks,
- $\mathcal{P} = \{(T_i, T_j) \mid T_i \prec T_j\}$ is a precedence relation that indicates which tasks depend on other tasks to be started,
- $\mathcal{D} = \{t_1, \cdots, t_n\}$ is the set of execution times of the n tasks. It is assumed that these times are deterministic if it were not for the interference of the interactive workload.
- $\mathcal{C} = \{c_{i,j} \mid (T_i, T_j) \in \mathcal{P}\}$ and $c_{i,j}$ is the communication demand between tasks T_i and T_j measured in number of bytes sent by T_i to T_j.

The precedence relation $\mathcal{P}$ is better depicted as a task graph where the nodes are the tasks of the parallel application and the arcs are elements of $\mathcal{P}$. Figure 1 shows an example of a task graph. The nodes of the graph have two labels: the top one is the task number and the bottom one is the task execution time. Arcs of the task graph are labeled with the communication demand. As seen in the figure, there are

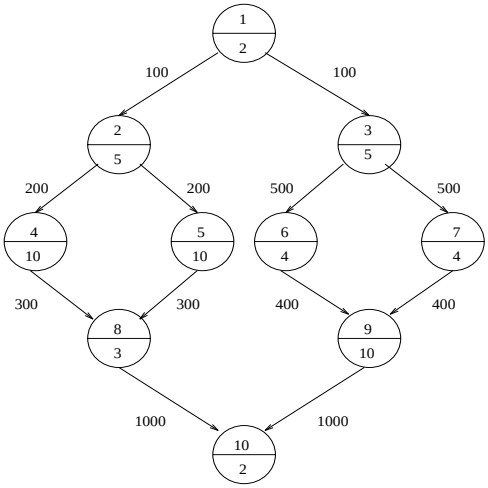

Figure 1: Task Graph Example

ten tasks in the parallel application. Task 1 is the initial task. Tasks 2 and 3 can only start when task 1 completes. Moreover, task 2 can only start when the data sent by task 1 is received by task 2 and task 3 can only start when it receives the data sent to it by task 1. Tasks 2 and 3 can be executed in parallel since they do not depend on one another. Tasks 4 and 5 depend on task 2 and tasks 6 and

7 depend on task 3. Task 8 depends on both 4 and 5 and task 9 depends on tasks 6 and 7. Finally, task 10—the final task— starts when both tasks 8 and 9 complete and when their data is made available to task 10.
We assume the existence of a task scheduler $\mathcal{S}$ that allocates tasks of a parallel program to any of the workstations. The criteria used by the scheduler may include workstation relative speeds and workstation loads by the owner.

3 Modeling Approach

Parallel tasks running on NOWs contend with two types of processes—owner jobs and other parallel tasks. Owner jobs have a higher priority over parallel tasks and hence run without suffering any delay. Parallel tasks thus experience some delay due to the presence of owner jobs. Concurrent parallel tasks running on the same workstation experience a performance degradation which is a function of the number of such tasks on that workstation. In this section we propose a model to predict the execution time of parallel applications. This model has an analytic based component and a simulation based component. Figure 2 shows the relationship between these components. In each iteration, the model

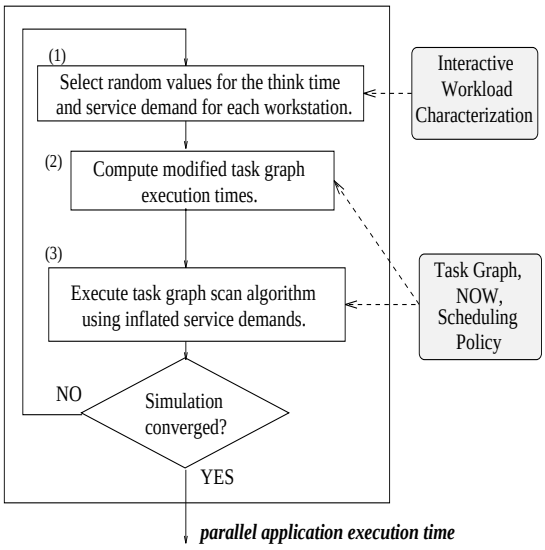

Figure 2: SimNOW performance model

computes the total execution time of the parallel application given a task graph, a NOW, a scheduling policy, and owner job information. The iteration is started by randomly selecting values for the think time $\tilde{Z}_i$ and cpu demand $\tilde{D}_i$ of owner jobs at each workstation i $(i = 1, \cdots, W)$, from an exponential distribution with average Z_i and D_i, respectively (see box 1 in Fig. 2). These values and the workstation in which each parallel task T_j will execute, according to the scheduling algorithm $\mathcal{S}$, are used to determine the modified (inflated) execution times, t'_j, for each task T_j, $j = 1, \cdots, n$, due to owner job's interference. Then, the actual execution time of each task is again modified to reflect the concurrent execution of other parallel tasks on the same workstation (box 2 in Fig. 2). These modified task execution times are then used to compute the execution times of the whole parallel application. This is accomplished by a task graph scan algorithm described in detail in subsection 3.3.1 (box 3 in Fig. 2).
This process is repeated as many times as needed until the computed values converge. The convergence criterium uses the batch means analysis technique to achieve a desired confidence level interval [7].
The next subsections describe in detail each component of the model.

3.1 Accounting for Owner Job Interference

As discussed before, owner jobs have preemptive priority at the CPU over parallel tasks. Therefore, parallel tasks only see the fraction of the CPU left unused by owner jobs. If U_i^{ow} is the CPU utilization due to owner jobs at workstation i, then the execution time t_j of a task T_j executed at workstation i should be inflated to a value t'_j as follows

$$t'_j = \frac{t_j}{1 - U_i^{ow}} \tag{1}$$

since the higher the CPU utilization of owner jobs, the longer it takes for parallel tasks to complete. To compute U_i^{ow} we need to remember that there are N_i owner tasks at workstation i, each with a think time of Z_i and CPU service demand of D_i. This can be modeled as an $M/M/1//N_i$ queuing system [3], i.e., a finite customer population single server system. The probability p_0^i of finding no customers in an $M/M/1//N_i$ system is given by [3]

$$p_0^i = \left[\sum_{k=0}^{N_i} \rho^k \frac{N_i!}{(N_i - k)!} \right]^{-1} \tag{2}$$

where $\rho = D_i/Z_i$. Then, the CPU utilization due to owner jobs is just $U_i^{ow} = 1 - p_0^i$.

3.2 Accounting for Interference from Other Parallel Tasks

All parallel tasks allocated to run on a given workstation will compete with one another for the CPU. The execution time of each one of them can be obtained by solving a multiclass closed queuing network (QN) model for the workstation. Each parallel task is assigned to a separate class on the QN model. Since we are not considering I/O, the number of devices in the QN model is equal to one (the CPU). Multiclass Mean Value Analysis (MVA) can be used to solve the QN model (see [9]). A workstation subscript will not be used in the equations that follow to simplify the notation. It should be clear however that equations 3 through 6 apply to a general workstation $i (i = 1, \cdots, W)$.

The MVA residence time equation for a parallel task of class r can be written as

$$R'_r(\overrightarrow{N}) = t'_r\ [1 + \overline{n}(\overrightarrow{N} - \overrightarrow{1_r})] \quad (3)$$

for $r = 1, \cdots, P_i$ where

- P_i is the number of parallel tasks assigned to workstation i,
- t'_r is the inflated (modified due to owner job's interference) execution time of a task r ($r = 1, \cdots, P_i$) allocated to workstation i,
- $\overrightarrow{N}$ is the population vector that represents the number of parallel tasks at the workstation for each class; since each parallel task is assigned to a different class, $\overrightarrow{N} = (1, 1, \cdots, 1)$,
- $\overrightarrow{1_r} = (0, 0, \cdots, 1, 0, 0)$ is a vector of zeroes except for the r-th position which is a 1, and
- $\overline{n}(\overrightarrow{N})$ is the total average number of parallel tasks at the CPU at workstation i as a function of the population vector $\overrightarrow{N}$.

The average number of parallel tasks at the workstation is simply the sum of parallel tasks at the workstation for all classes. Thus,

$$\overline{n}(\overrightarrow{N}) = \sum_{r=1}^{P_i} \overline{n}_r(\overrightarrow{N}) \quad (4)$$

Since, $\overline{n}_r(\overrightarrow{N}) = 1$ for all $r = 1, \cdots, P_i$ we have that,

$$\overline{n}_r(\overrightarrow{N} - \overrightarrow{1_r}) = P_i - 1 \quad (5)$$

Combining eqs 3 and 5 we get

$$R'_r(\overrightarrow{N}) = t'_r\ [1 + P_i - 1] = P_i\ \ t'_r \quad (6)$$

Thus, the execution time of a parallel task of type r at a workstation is equal to the number of parallel tasks executing on the workstation multiplied by the task's inflated service demand.
If we combine equations 1, 2, and 6 we obtain the execution time t^i_j of task T_j when assigned to workstation i

$$t^i_j = P_i\ t_j\ \left[\sum_{k=0}^{N_i} (D_i/Z_i)^k \frac{N_i!}{(N_i - k)!}\right] \quad (7)$$

3.3 Computing Parallel Task Execution

In this subsection we discuss the Task Graph Scan (TGS) algorithm used to compute the execution time of the parallel application. A few definitions and notation are in order. Let,

- succ(T_i): set of tasks that are successors of T_i.
- pred(T_i): set of tasks that are predecessors of T_i.
- start(T_i): start time of T_i
- finish(T_i): finish time of T_i
- status(T_i): status of parallel task T_i. It can be
  - NA: task is Not Assigned to a workstation or has not been started
  - EX: task is executing, or
  - FD: task has finished execution.
- WS(T_i): workstation assigned to T_i.
- $\mathcal{X}$: set of executing tasks,
- set of ready tasks, i.e.
  $\mathcal{R} = \{T_j \mid \text{status}(T_j) = \text{NA}\ \&\ \forall\ T_k \in \text{pred}(T_j), \text{status}(T_k) = \text{FD}\}$

The following functions are required by the algorithm TGS.

- The function startdelay() returns the maximum startup delay for a parallel task. This delay is due to the predecessor of the task which has the largest execution time. So,
  $\text{startdelay}(T_j) = \max_{T_i \in \text{pred}(T_j)} \{\text{finish}(T_i) + c_{i,j}\}$
- W(T_j, $\mathcal{S}$): workstation assigned to T_j by scheduling policy $\mathcal{S}$.
- SD(T_j, w) = t^w_j: execution time of T_j running on workstation w. This is computed using eq. 7

3.3.1 Algorithm Description

The algorithm starts by initializing the start times of all tasks and setting the status of all tasks to NA. Then, the first task in the application, T_1, is scheduled to a workstation using the function W(). The execution time for T_1, including the inflation due to interactive jobs, is computed. The finish time for the initial task is simply its execution time.
The algorithms then loops until all tasks are marked as complete. In each loop iteration, three procedures are executed: ComputeNextSet, StartTasks, and FinishTasks. The ComputeNextSet procedure computes the set $\mathcal{N}$ of tasks to be started next. This set is a subset of the set $\mathcal{R}$ of ready tasks and consists of all tasks in $\mathcal{R}$ that have the minimum start delay. These are the first tasks to be scheduled. The StartTasks procedure sets the status of all tasks in $\mathcal{N}$ to EX to indicate they are executing, include them in the set $\mathcal{X}$ of executing tasks and computes their finish time. Finally, procedure FinishTasks compute the set $\mathcal{F}$ of the first tasks to finish. These tasks are removed from the set $\mathcal{X}$ and their status is marked as finished.
The TGS algorithm can be formally specified as follows:

/⋆ **Initialization** ⋆/
status(T_i) ← NA $\forall\ T_i \in \mathcal{T}$; start(T_1) ← 0
$WS(T_1)$ ← $W(T_1, S)$;
finish(T_1) ← SD(T_1, WS(T_1)) ; status(T_1) ← FD
/⋆ task scan ⋆/
Repeat
    ComputeNextSet();
    StartTasks();
    FinishTasks();
Until (status(T_k) = FD $\forall\ T_k \in \mathcal{T}$)

The procedures ComputeNextSet, StartTasks, and FinishTasks are given below:

Procedure ComputeNextSet();
    $\mathcal{N} \leftarrow \{T_k \mid T_k \in \mathcal{R}$ & startdelay(T_k) = $\min_{T_j \in \mathcal{R}}\{\text{startdelay}(T_j)\}$
End Procedure;

Procedure StartTasks();
    For each task T_k **in** $\mathcal{N}$ **do**
        status(T_k) ← EX
        $\mathcal{X}$ ← $\mathcal{X}$ ∪ $\{T_k\}$
        start(T_k) ← startdelay(T_k)
        WS (T_k) ← W ($T_k, \mathcal{S}$)
        finish(T_k) ← start(T_k) + SD (T_k, WS(T_k))
    End do
End Procedure;

Procedure FinishTasks();
    $\mathcal{F}$ ← $\{T_k \mid T_k \in \mathcal{X}$ & finish(T_k) = $\min_{T_j \in \mathcal{X}}\{\text{finish}(T_j)\}\}$
    For each task T_k **in** $\mathcal{F}$ **do**
        status(T_k) ← FD
        WS(T_k) ← Φ
        $\mathcal{X}$ ← $\mathcal{X}$ − $\{T_k\}$
    End do
End Procedure;

Table 1: Task Graph Scan Algorithm

4 Model Validation

The performance prediction model was validated through measurements taken on a network of SUN workstations at the Center for the New Engineer. The application used for the validation studies is illustrated in the task graph of figure 3. This application has an increasing degree of

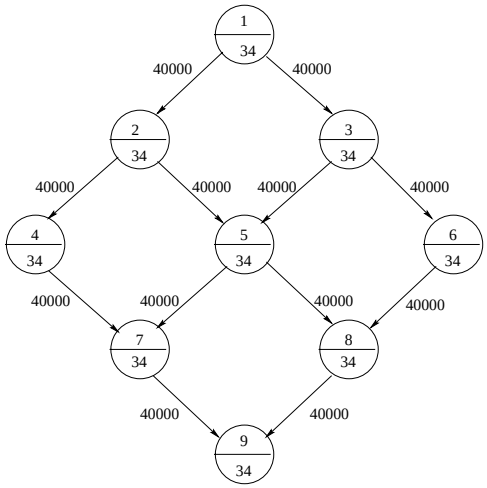

Figure 3: Task Graph for Validation Experiment

parallelism (from 1 to 3) followed by a decreasing degree of parallelism. Some tasks have two successors, some have one successor, and one task (the final task) has no successor tasks. Some tasks have two predecessors, some have one predecessor, and the initial task has no predecessor. This parallel application was chosen for the validation experiment because it exhibits a rich pattern of task interdependency and is typical of wave-front computations. The application was implemented using PVM (Parallel Virtual Machine) [10] which is an integrated set of software tools and libraries that emulates a general-purpose concurrent computing framework on interconnected computers of different architectures. All tasks in the task graph of figure 3 run the same computation—a matrix multiplication—and use PVM's send and receive primitives to send and receive 40,000 bytes to and from other tasks. The computation time for each task is approximately 34 seconds on a SUN LX workstation.

The validation consisted in running the parallel application on a network of workstations with different numbers of workstations and measuring the total time, T_{meas}, taken to execute the application. This result was then compared with the time T_{mod} obtained from our model for the same application and NOW configuration. The percent relative error E was computed as

$$E = \mid \frac{T_{meas} - T_{mod}}{T_{meas}} \mid \times 100.0 \qquad (8)$$

Two different experiments were conducted. The first consisted in running the parallel application of figure 3 with no interactive workload in any workstation. The results of the validation are shown in table 2. The table shows the results of several measurements obtained by running the application with three different types of workstations—Sun LX, Sun SPARC5, and SUN SPARC20—and a number of workstations varying from 1 to 3. The table shows the total execution time (in seconds) of the parallel application as a function of the number of workstation and the type of workstations used. Since the application has a maximum parallelism of 3, contention for the workstation occurs for 1 and 2 workstations. The same type of workstation was used in each experiment to run all tasks. The table shows three sets of results columns, one for each type of workstation. Each set presents results obtained from the measurements, from the model, as well as the relative error E. We do not show measurements for 2 and 3 SUN SPARCstation20's since we only have one of these in our lab.

As can be seen from table 2, the errors between the model predictions and the measurements range between 2.9 and 11%.

The second set of validation experiments were aimed at validating the part of the model that takes into account the interference of the workstation owner workload. For that purpose, we developed a daemon that emulates the load that would be placed on a workstation by a command submitted by an interactive user. This daemon process sleeps for Z seconds on the average—to simulate the think time—and awakes to perform a computation that takes D sec to simulate the computation demand of the interactive workload. After that, the daemon goes back to sleep. N instances of the daemon may co-exist in the system.

Table 3 shows the results obtained for the validation that considers the interactive workload. All experiments were run at night to insure that no other user workload, except our synthetic user workload, was present in the system. The results shown in the table consider that there are two daemons running at each workstation ($N = 2$), each one uses 1 sec of CPU each time it awakes (D =1 sec), and the sleep time is equal to 5 sec ($Z = 5$ sec). These are going to be our baseline parameters for the interactive workload for the performance prediction studies considered in the next section. As before, we can see that errors are fairly small and are in the range between 4.6 and 8.4 %.

5 Performance Prediction

Once the model is validated, it can be used to conduct all sorts of performance prediction studies. This section describes three such studies that show how different aspects of the owner workload interfere with the execution time of the parallel application. The parallel application is the same as the one used for the validation experiments. The parameters Z, D, and N for the owner workload are assumed to have the baseline values unless stated otherwise. We assume that three workstations were used to obtain all

No. of Workstations	Sun LX			Sun SPARCstation 5			Sun SPARCstation 20		
	T_{meas}	T_{mod}	E	T_{meas}	T_{mod}	E	T_{meas}	T_{mod}	E
1	300.576	309.320	2.9	145.915	137.540	5.7	113.571	105.837	6.8
2	223.284	204.320	8.5	103.037	91.800	11.0	-	70.665	-
3	204.800	192.984	6.2	88.557	86.717	2.1	-	66.756	-

Table 2: Ideal Case (No Owner jobs) Results

No. of Workstations	Sun LX			Sun SPARCstation 5			Sun SPARCstation 20		
	T_{meas}	T_{mod}	E	T_{meas}	T_{mod}	E	T_{meas}	T_{mod}	E
1	432.331	453.200	4.6	217.804	205.828	5.5	167.342	156.486	6.5
2	327.670	302.240	7.7	150.048	137.425	8.4	-	104.403	-
3	302.695	285.433	5.7	117.506	127.537	7.9	-	99.936	-

Table 3: Owner Job Interference Case Results

the plots in this section.
Figure 4 shows the variation of the execution time of the parallel application versus the service demand D of the owner workload for various types of workstations.

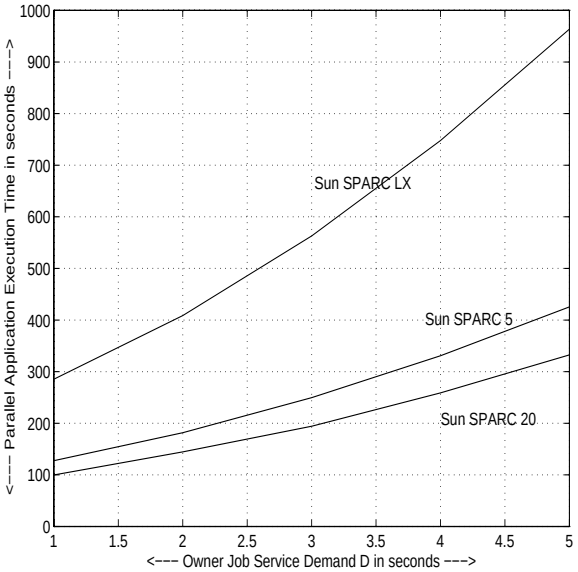

Figure 4: Execution Time versus interactive load computing demand

As it can be seen, the computing demand of the interactive workload has a non-linear effect on the execution time of the parallel application. Slower workstations suffer a bigger impact than faster ones.
Figure 5 shows the variation of the execution time of the parallel application versus the number N of concurrent owner submitted commands for various types of workstations. The curves show a similar kind of degradation as the one seen in figure 4. For example, at $N = 5$ the execution time T is 2.9 higher using LX workstations than using SPARCstation 20's. However, at $N = 1$, the degradation factor is 2.65. As N increases, the slower workstations show an increasing degradation factor when compared with the faster ones.

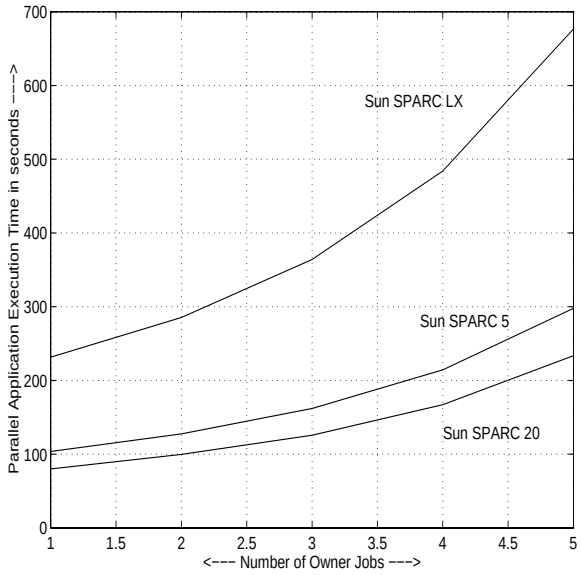

Figure 5: Execution Time versus Number of Owner Jobs

Finally, figure 6 shows the variation of the execution time of the parallel application versus the think time Z of the owner workload for various types of workstations. The figure shows the dramatic decrease in the parallel execution time of the parallel application as the think time increases, which amounts in a decrease in the workload intensity of the load imposed by the owner.

6 Brief Description of SimNOW

In this subsection we briefly describe the Graphical User Interface of SimNow—a tool developed by the authors to predict the performance of parallel applications running on networks of workstations. The model used in SimNOW was

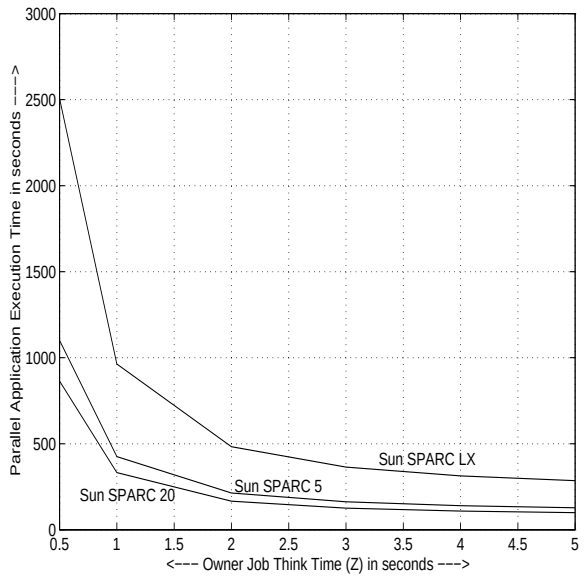

Figure 6: Execution Time versus Owner Think Time

described in the previous sections.
A user friendly GUI was developed using OSF/Motif 1.2. Motif was chosen as the GUI development tool over OpenLook to aid in cross platform compatibilities. Another factor was the availability of the tool in our lab. The main features of the GUI are:

- User friendly and intuitive components: every attempt was made to make the components, such as pushbuttons and menus, as intuitive as possible.
- Interaction through modal dialogs prevent the user from making mistakes during the configuration of the task graph. Also, model validation of all data entered is made through the GUI rather than the back end (the model solver). This isolates error checking from normal execution of the back end.
- A WYSIWYG task graph editor helps easy creation of the task graphs.
- One to one correspondence between GUI objects and the model components.

A snapshot of the SimNOW Graphical User Interface is shown in figure 7. The top horizontal menu has five submenus: File, Edit, View, Run, and Help. The File Menu has options to start a new model, load an existing model, save an open model, close an open model, and quit the tool. Once a model is open, a graphical editor can be used to specify the task graph for the parallel application (see task graph example in figure 7 in the task graph canvas). The buttons labeled Node and Link above the canvas can be used to add nodes to the task graph and link tasks to one another.
The Edit menu is used to edit the task graph being constructed. Its options allow the user to move objects (tasks and links), delete objects, and clear the drawing area. The View menu has options to view the model inputs in a scrolling text area in the lower left portion of the main screen, clear inputs, and clear output screens. The Run menu has options to either execute the model or just validate it.
The scheduling policy option menu above the task graph editor canvas (see figure 7) enables the user to select a scheduling policy from the following types of policies

- PVM DEFAULT: the default scheduling policy used in PVM [10]. Parallel tasks are allocated to workstations in a round-robin fashion regardless of the type of workstation or the interactive load. Typically, under this scheduling policy, at the beginning of the parallel program, or as new tasks are spawned, the PVM scheduler allocates the task to the next workstation in its list of hosts. The advantage of such a scheduling process is that the overhead due to allocation is very small. However, this scheduling policy is very inefficient from the parallel application point of view. The scheduling disciplines discussed below try to utilize the computing resources in better ways.
- Workstation Speed Dependent Scheduling: this is a modification over the default PVM scheduling policy. In this policy, parallel tasks are allocated to faster workstations first and then to slower ones. This policy generates poor scheduling when the initial tasks have smaller computational demands than the ones that follow them because parallel task demands are not considered.
- Parallel Load/Speed Scheduling: this scheduling policy tries to account for both workstation speeds and parallel task computational demand before allocating a task to a workstation. In this policy, workstations are picked in the order of their speed with faster workstations chosen for allocation before slower ones. Then, the parallel task with the largest computation demand is allocated to a faster workstation.
- Workstation Load/Speed Scheduling: this scheduling policy is an improvement over the previous policy in that the current load at the workstation is taken into account along with the computing demand of the parallel tasks and the workstation speeds. The policy works as follows:

  1. Determine the total service demand at a workstation (existing + the proposed task demand).

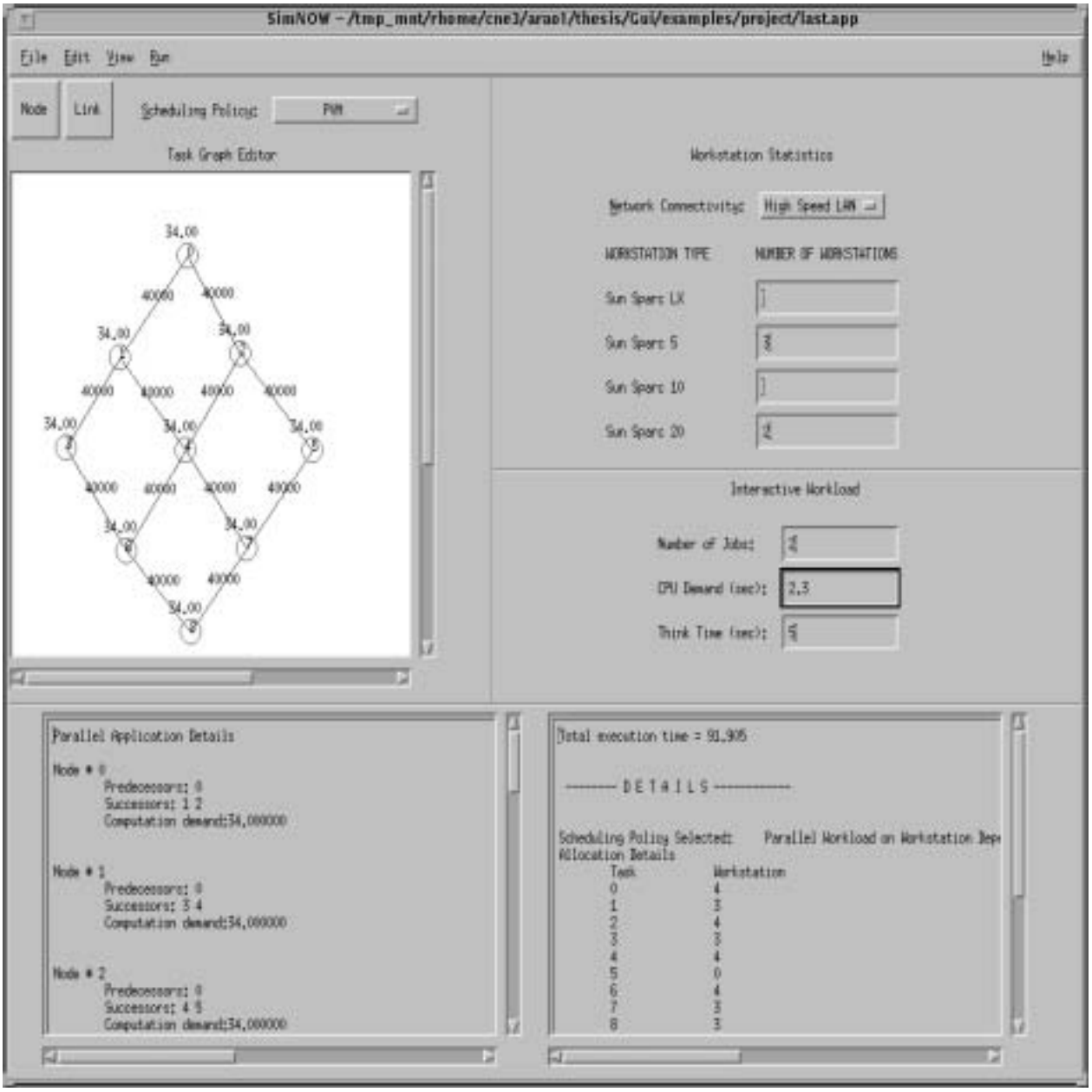

Figure 7: Main Screen of the Tool

2. Find α—the ratio of the total service demand to the workstation relative speed. Allocate the parallel task under consideration to the workstation for which α is the least.

This policy uses the concept of "Ready Queue" of parallel tasks. The ready queue is the queue in which all currently schedulable parallel tasks are placed. The scheduler then takes these tasks one after the other and allocates them to workstations. A task can be in the ready queue if and only if all its predecessors have completed execution. At this point it is put in the ready queue from where the parallel task scheduler decides which workstation to execute the parallel task on. The decision is based on the value of α.

Users can select the type of network that connects the workstations using a pull down menu (see figure 7). The available types are 10 Mbps Ethernet LAN, 4 and 16 Mbps Token Ring, 100 Mbps FDDI ring, and 155 Mbps ATM LAN.
Right below the network specification menu, the user can provide the number of workstations of each type used to configure the NOW. Below this area, the user specifies the workload parameters (number of jobs, CPU demand, and think time) for the interactive workload.
After the model is solved, results are shown in a scrollable text window as shown in figure 7.

7 Conclusions

Many networked environments have significant idle capacity that can be used to run tasks of parallel applications. The performance of parallel applications running on networks of workstations depend on several issues including the interference of the interactive workload submitted by the owners of the workstation, the parallel applications characteristics, the scheduling algorithm used to allocate tasks of the parallel application to workstations, the mix of workstations used to run the application, and the type of network used to connect the workstations. This paper presents a model that allows designers of parallel applications to predict the their performance taking into account all of the issues just mentioned. The model was validated through measurements and proved to be very accurate. To make the model easier to use, a tool that implements the model was developed. It has a graphical user interface through which users can specify a task graph using a graphical editor, run the model, and view the performance results.

References

[1] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW team, "A Case for NOW (Network of Workstations)", Proceedings of the Principles of Distributed Computing Conference, August 1994.

[2] Phillip Krueger and Rohit Chawla, "The Stealth Distributed Scheduler", II International Conference on Distributed Computing Systems, 1991.

[3] Leonard Kleinrock, *Queuing Sytems, Volume I: Theory*, John Wiley, New York, 1975.

[4] J. C. Little, "A proof of the queuing formula $L = \lambda W$," Operations Research, Vol. 9, pp. 383–387 (1961).

[5] Michael J. Litzkow, Miron Livny, and Matt W. Mutka, "Condor - A Hunter of Idle Workstations", Proc. of the 8th International Conference on Distributed Computing Systems, IEEE, June 1988.

[6] Allan Bricker, Michael Litzkow, and Miron Livny, "Condor Technical Report", TR 1069, Computer Sciences Department, University of Wisconsin-Madison, January 1992.

[7] MacDougall, M.H., *Simulating Computer Systems: Techniques and Tools*, MIT Press, Cambridge, MA, 1987.

[8] Marvin M. Theimer, Keith A. Lantz, "Finding Idle Machines in a Workstation-Based Distributed System", IEEE Transactions on Software Engineering, November 1989.

[9] Menascé, Daniel A., Virgilio A. F. Almeida, and Larry W. Dowdy, "Capacity Planning and Performance Modeling: from mainframes to client-server systems," Prentice Hall, Englewood Cliffs, NJ (1994).

[10] Geist, A., A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam, "PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing", MIT Press, Cambridge, MA, 1994.