Providing Resource Management Services to Parallel Applications*

Jim Pruyne† Miron Livny†

Abstract

Because resource management (RM) services are vital to the performance of parallel applications, it is essential that parallel programming environments (PPEs) and RM systems work together. We believe that no single RM system is always the best choice for every application and every computing environment. Therefore, the interface between the PPE and the resource manager must be flexible enough to allow for customization and extension based on the environment. We present a framework for interfacing general PPEs and RM systems. This framework is based on clearly defining the responsibilities of these two components of the system. This framework has been applied to PVM, and two separate instances of RM systems have been implemented. One behaves exactly as PVM always has, while the second uses Condor to extend the set of RM services available to PVM applications.

1 Introduction

To fulfill the promises of high performance computing, parallel applications must be provided with effective resource management services. The Resource Manager of a system determines how long an application must wait for the CPUs it requires, how much memory will be allocated to it, and how balanced the computation will be. The resource manager is responsible for allocating resources among all jobs submitted to the system (Inter-Job RM), and binding resources to requests made by a single job (Intra-Job RM). Without effective inter-job RM, resources may be misallocated among users, placing some at an unfair disadvantage. Poor intra-job RM may cause users' programs to perform poorly because requests may be fulfilled using resources which are heavily loaded or do not closely match the application's requirements.

A number of specialized batch resource management systems are currently available. Systems of this type include Condor [1], DQS [2], and IBM's LoadLeveler. To use these systems, a user submits a job description file which specifies the program to be run, and the type of resource on which to run it. The RM system uses this description to find a suitable host on which to run. In addition, the RM system starts and monitors the job, and informs the user when it completes. Systems of this type have been used successfully in handling serial jobs, but they generally have not been adapted to support parallel jobs effectively.

Parallel applications make resource management more complicated. Inter-job RM is complicated by the necessity of allocating groups of possibly heterogeneous hosts to a single application. This decision may involve, for example, choosing between allocating a few resources to a number of jobs with small demands, or providing many resources to a

*This work was partially funded by a research grant and fellowship from IBM.

†Dept. of Computer Sciences, University of Wisconsin–Madison

single job with large demands. Parallel applications also require Intra-Job RM services. For example, the RM system must start processes as they are requested by the user. These processes should be distributed so that each process is executed on appropriate hardware and so that the load across all machines remains balanced. Finally, unlike sequential applications, parallel applications may be able to make use of new resources if they become available during the run. A resource manager for parallel applications must therefore continue to interact with the application during the course of the run to make changes in the state of the global system visible to running jobs.

The variety of parallel programming environments which are available pose additional complication in doing resource management for parallel applications. To get applications started, these systems generally provide simplistic RM functionality, but they are not as flexible as specialized RM systems. A specialized RM system is capable of providing additional functionality beyond what the parallel programming environment provides, but interfacing the parallel programming environment and the RM system currently requires extensive modifications to both systems.

To allow existing RM systems to be more easily integrated with parallel programming environments, we propose that all RM functionality be removed from the PPE code, and migrated into one or more processes which are dedicated to handling resource management requests. RM processes receive requests directly from an application using the same primitives which are used for communication among the tasks comprising a user's parallel application. This approach has a number of advantages. First, adapting a new RM system to a PPE does not require any changes to the PPE code. Second, an RM system may customize or extend the set of RM functions defined by the PPE. Finally, a single RM system does not need to be changed greatly to support new PPEs. Only the communication primitives used need to be changed.

The remainder of the paper is organized as follows. The next section describes a framework in which resource management requests are handled by tasks external to the PPE. Section 3 describes changes we made to PVM to support this architecture, and the development of an external RM task which mimics stand alone PVM. Our experience using the Condor distributed batch system as a PVM resource manager is detailed in section 4. Concluding remarks and future directions are presented in section 5.

2 Handling Run-Time Resource Management Requests

The first step in migrating resource management services out of a PPE is determining exactly which services will be handled by the RM processes and which will continue to be handled by the PPE. We view the resource manager as the decision maker in the system. Therefore, all service requests pertaining to host addition or deletion, task assignment, and queries about the state of the system should be in the domain of the resource manager. The PPE is responsible for communication and other services which help insulate the application and the RM from the underlying operating system and hardware. The PPE must be careful not to provide application tasks with access to services which circumvent the policies imposed by the RM. For example, process creation is a desirable service for the PPE to provide, but this service should only be available to RM processes. By handling process creation, the PPE shields the RM developer from the diverse process creation primitives supplied by various operating systems. However, if application tasks are able to create processes directly they may corrupt the efforts of the RM to maintain load balancing. Figure 1 shows the layering of service providers.

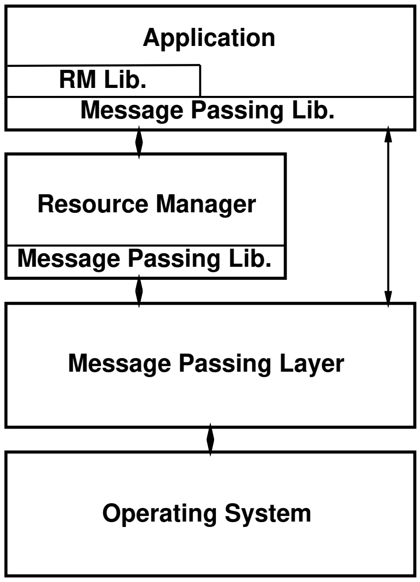

Fig. 1.

The distinction between resource management services and communication services is enforced by splitting them into separate libraries. The communication library contains only an interface to the communication primitives supported by the PPE. Communication among tasks comprising a parallel application is accomplished using this library directly. The RM library uses the communication library to send requests for service to the RM task associated with the process. RM processes also use the communication library to access services provided by the PPE for communicating with application tasks.

2.1 Handling Run-Time Intra-Job Resource Management Requests

Figure 2 shows a generic Resource Management request exchange between an application task and the RM process associated with it. We believe that all run-time RM requests should be handled asynchronously. That is, when a process makes a request for a RM service, it should not be blocked waiting until the request is fulfilled. Making RM requests asynchronous has a number of advantages. First, the time required to fulfill a RM request may be significant. For example, a request to add a new host to a job may take a very long time if all of the machines in the system are in use. Blocking the application for this length of time is unacceptable. Second, asynchronous handling of requests allows for more parallelism by permitting a task to have more than one RM request outstanding at a time. This is especially advantageous is in starting a group of new processes. Instead of making a series of requests to start individual processes, an application can have a number of start process requests outstanding at one time. These requests may be handled in parallel by the RM reducing the overall time required to start the processes.

Figure 2 also shows the format for general RM request and response messages. The request message contains not only the type of the request and data associated with the request, but also request ID and response type fields. The request ID field is the key to asynchronous handling of RM requests. The RM library generates a unique request ID for each request made by an application task, and returns its value to the caller immediately. The request ID is then passed to the RM task as part of the request message. When the

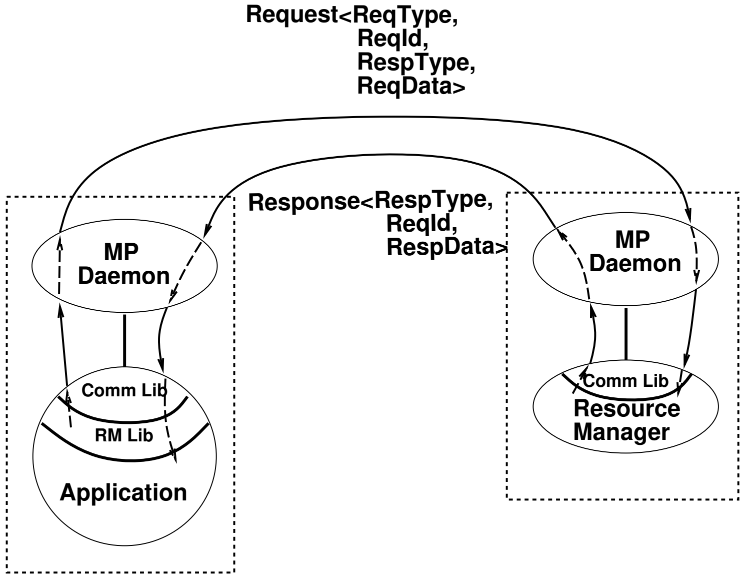

FIG. 2.

resource manager composes a response to a service request, it includes the request ID. The application task then uses the request ID field in the response message to match the response with an outstanding request. By using unique request IDs, a single application task can differentiate between responses for any number of outstanding requests, even if the requests are for the same type of service. The request message also contains a field specifying the desired message type to be used by the RM when sending a response to this request. The system may provide a default value for this field based on the type of request if the application writer does not specify a value when making a request.

3 Resource Management for PVM

Our first implementation of the external resource manager architecture has been done using PVM [3]. PVM was chosen for a number of reasons. First, the PVM source code is available which makes experimentation possible. Second, PVM is widely used, and we hope that our work will benefit as many users as possible. Finally, unlike many other parallel programming environments, the PVM application programming interface handles dynamic addition and deletion of hosts and processes. This provides us with a basic set of primitives to implement in the RM rather than having to define new primitives before gaining experience with the system. A very gratifying aspect of our use of PVM has been the relationship we've developed with the PVM development team at the University of Tennessee. They have been very receptive to our ideas, and the changes described below became a part of PVM as of release 3.3.

Prior to release 3.3, all requests for RM services were handled by a combination of the PVM library and the PVM daemons. In general, an application made a request by calling a function in the PVM library. The library translated the request into a message to the PVM daemon on the same host. The PVM daemon handled the request locally when possible (for example a pvm_spawn() request where the new process will be started on the same host), or forwarded the request to a remote PVM daemon which it believed will be able to handle the request best. The remote PVM daemon then carried out the request and sent

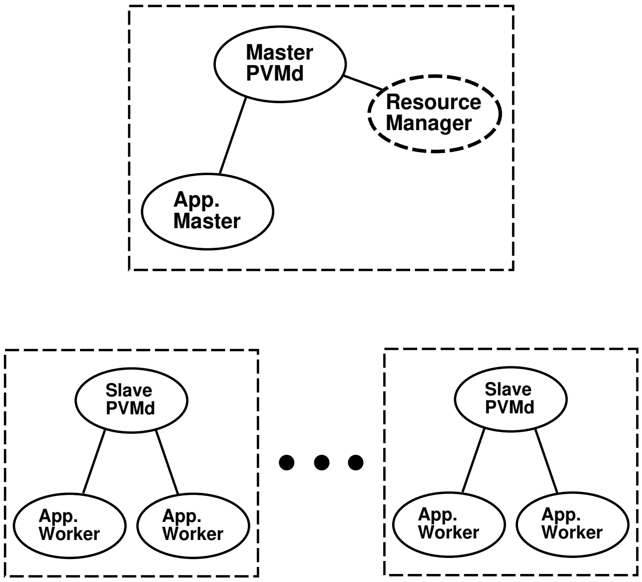

Fig. 3.

a reply back to the daemon local to the requesting task. The local daemon completed the loop by forwarding the the response to the requesting task.

To move the handling of Resource Management requests outside the PVM daemons, the first step was simply to introduce the notion of resource manager tasks. Each PVM daemon associates itself with a RM task. This RM task may or may not be on the same host as the daemon. Figure 3 shows the most basic RM configuration possible in which there is only one RM task in the system which is responsible for all of the hosts. When new hosts are added, the PVM daemon on the new host inherits the RM task of the master PVM daemon.

PVM daemons will perform some services for RM tasks which they will not perform for any other tasks within the system. The most important RM only service is process creation and monitoring. When a RM task chooses a host on which to run a new process, it sends a message directly to the PVM daemon on that host. The daemon responds by starting the process, and reporting the status of the new process to the RM. When a process exits, the PVM daemon sends a message to the RM task containing the exit status and the resource usage of the process. This scheme helps to insulate the RM task from the underlying operating system. No matter what OS is running on a host, the RM can start a task and receive status information simply by sending and receiving messages. As more and more types of hardware (such as massively parallel machines) come to be supported by PVM this service becomes increasing useful in maintaining generality in resource managers.

As part of the process creation service, the PVM daemon passes the identity of its RM task to the new processes. All future requests for resource management services made by the new task will be sent to its RM task. An application may discover the identity of its RM task by calling the new PVM library function get_rm_id(). As in the general architecture described above, the RM portions of the PVM library have been separated into a new RM specific library. The RM library uses get_rm_id() to know where to send requests for service. Functions have been added to the PVM library which set and retrieve a new request identifier field in the message header. Because all PVM RM requests are synchronous, this field is not strictly required for normal PVM operation, but as new, asynchronous primitives are provided by more complex resource managers, it will be needed to match responses with requests.

RM tasks initialize themselves with the PVM system via a new function called pvm_reg_rm(). At system start-up, a PVM daemon is not associated with any RM task, so it will simply adopt any task which calls pvm_reg_rm() as its RM. After a PVM daemon has been associated with an RM task, other tasks which call this function must be validated by the current RM task. This scheme permits a flexible topology of RM processes, but helps to reduce the risk of a malicious task gaining RM privileges.

3.1 The Vanilla PVM Resource Manager

As a simple test of the external RM concept, we have developed a "vanilla" resource manager task which mimics the behavior of stand alone PVM. The vanilla RM is integrated with the PVM console program. Therefore, when a user starts the console program to begin a PVM session, they also start the RM for the system. The vanilla RM accepts standard PVM host files, and initializes hosts in the file just as if the host file was passed to the master PVM daemon at start-up. Because the vanilla RM implementation was meant to be simple, it never grants a request for another task in the system to become a RM task.

After all of the PVM modifications described above were in place, development of the vanilla resource manager was straightforward. In most cases, the functions in the PVM daemon code which handled an application request for service could be moved into the RM code with few modifications. The needed changes were generally cosmetic, and did not require extensive algorithmic changes.

The goal of the vanilla resource manager is not to improve the performance of PVM, but simply as proof of concept. We also hope that by providing a simple RM, others will be able to modify or replace it to suit their application and environment. None the less, a few interesting performance changes were noted. These changes are due to the centralized decision making of the RM as opposed to the distributed decision making implemented in the PVM daemons. The area in which this is most dramatic was load balancing. The RM's centralized information allowed it to place new processes on the host with the smallest number of PVM processes. Because hosts may have load from sources completely outside of PVM, this does not insure an optimal distribution of tasks, but it does seem superior to the distributed decision making done by the PVM daemons which may place many tasks on a single host while leaving another host completely unloaded. RM requests such as pvm_tasks() which has to gather information from all hosts in the system are significantly faster due to the centralized information at the RM. On the other hand, requests such as pvm_config() which can always be carried out by a task's local daemon pay a penalty for accessing a remote RM task.

4 Condor as a PVM Resource Manager

Condor is a batch resource management system for workstation clusters. It schedules jobs submitted to the Condor system on idle workstations within the cluster. When an owner reclaims a machine, Condor withdraws the job and reschedules it on another idle workstation. Condor's ability to manage and schedule jobs on large number of workstations make it an ideal candidate for an inter-job resource manager for parallel applications as well.

4.1 Advantages of Using the Condor RM

In stand alone PVM, users have been required to explicitly select machines on which to run their programs. They cannot be expected to monitor the load on all of the machines available to them and always pick only lightly loaded machines. Unfortunately, choosing a heavily loaded machine may cause it to become a bottleneck for the entire parallel computation. Condor's guarantee of unloaded machines insures users that no outside influence will perturb the performance of their program. This not only improves program performance, but helps to make a parallel program's performance more consistent from run to run. Consistent performance results help application writers tune their programs by insuring that bottlenecks are inherent to the application and not due to some outside influence.

Perhaps Condor's greatest benefit to users is the large number of machines to which it can provide access. PVM has traditionally used the services provided by the operating system to access machines in a pool. This usually means that users must have accounts on every machine on which they will run. Condor allows users' jobs to run on any machine in the Condor pool regardless of whether the user submitting the job has an account there or not. This may greatly expand the number and type of resources a user has access to.

4.2 Changes to PVM Visible to Applications

The advantages Condor provides to users require that some changes be made to how PVM looks to an application. Because Condor selects the machines on which a job will run, all dependencies on individual host names must be removed from the application. In their place, the application must use class names. Each machine belongs to exactly one class, and all machines within a class are considered to be equivalent for the purposes of resource management decisions. The attributes of hosts within a class are defined in the job description file the user submits to Condor. Attributes must include the processor type and operating system for hosts in the class, but may also include characteristics such as available real memory, disk and swap space. These parameters help the user to insure that a process will not fail due to insufficient disk or swap space, or perform poorly due to lack of real memory. All requests for new hosts or starting new processes require that the user specify the desired class rather than the desired host as is usually done with PVM.

Because owners may reclaim their machines at any time, "host failure" becomes a relatively frequent occurrence. When this happens, Condor kills all tasks running on the machine. An application can detect this event using the often overlooked pvm_notify() mechanism. This mechanism allows a user's task to be informed of host and process exit due to events like Condor reclaiming a machine. It is up to the application, however, to deal with the host loss.

When an application requests that a new host be added to the computation, Condor attempts to schedule one from the pool of idle machines. This process requires a significant period of negotiation and therefore may take several minutes. Additionally, if there are no machines of the desired type currently available, or the user making the request has low priority, the delay may be even longer. To avoid forcing the application to block while this request is being carried out, requests for new hosts, as well as requests for host deletes (which may also take considerable time), are handled asynchronously. The application uses the pvm_notify() mechanism to detect when one of these asynchronous requests has completed.

Finally, when Condor first detects activity by a workstation owner, it suspends all

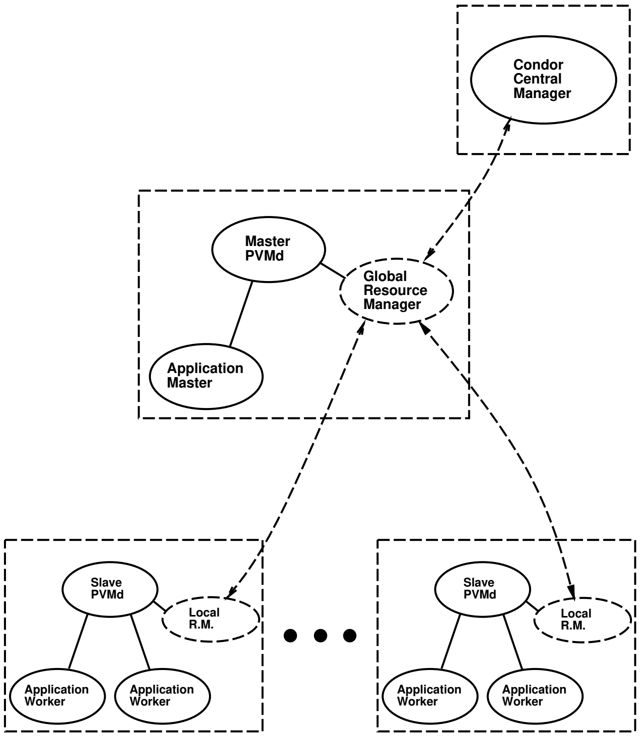

FIG. 4.

processes running there rather than killing them immediately. If the owner remains active for less than ten minutes, Condor allows the processes to resume. This saves the overhead involved in replacing a host when an owner is only active for a short time. A parallel application may, however, need to know that a process has been suspended. This is particularly useful if there is synchronization with the suspended task which requires a number of other tasks to block. To help the application in dealing with this situation, the pvm_notify() mechanism has been extended to inform the application of host suspends and resumes.

4.3 Architecture of PVM with Condor RM

Figure 4 shows the architecture of PVM when Condor assumes the roll of resource manager. There are a number of differences between this figure and figure 3 which showed the most simple topology of RM tasks possible. First, note that each PVM daemon has a separate RM task associated with it. On the slave machines, the Condor process which is responsible for monitoring a job running on a machine acts as the RM. It uses PVM to start tasks, send signals to suspend, resume and kill tasks, and receives process completion information from the daemon. The local RM tasks communicate with the global RM which is located on the machine where the job is submitted to keep the global RM up to date on the status of the machine, and to forward all requests for resource management services made by local tasks. The global RM runs on the machine on which the job was submitted to Condor. This is generally the home machine of the user, and considered to be stable for the life of the run. It therefore also holds the master PVMd and the initial process of the user's application. The global RM handles all user requests for RM services either directly sent from the application master, or indirectly forwarded by a RM task local to a slave process.

When requests are made for new hosts to be added, the global RM task communicates with the Condor central manager to schedule a new machine. When the machine is granted, the global RM insures that it is configured and then informs any application tasks which have requested notification of host addition.

4.4 Experience with Condor as a PVM Resource Manager

We have a running prototype using Condor as a PVM resource manager. To test the system, we have developed an application shell which attempts to acquire new hosts as often as possible. When a new host is received, a worker process is spawned there which repeatedly loops for a number of iterations specified by the application master. When the loop iteration count is completed, the worker reports its result along with the amount of CPU time it consumed, and then goes back to looping. Table 1 shows the results of one long execution of this application. From this table, we see that the job was able to accumulate nearly 53 times as much CPU time as elapsed wall clock time. The last three columns of the table require some explanation. The only RM service which is not handled asynchronously in our system is spawning a new process. While a spawn request is outstanding, the application master process is blocked. The spawn occupancy column reports the product of the total time the application master was blocked and the number of machines running while it was blocked. This gives a worst case amount of CPU time lost due to the blocking. As mentioned previously, Condor first suspends hosts for up to ten minutes before removing them from a computation. The suspended time column shows that hosts allocated to the computation spent slightly more than 15 hours in the suspended state. Accounting for the time lost due to suspends, the job was able to utilize 97last column shows that over the course of the run, the job was scheduled on 161 different hosts. This number corresponds to the number of machines which were reclaimed by owners during the run. An important result which table 1 does not show is that when a machine is lost, a replacement was typically obtained from Condor and ready to begin computation less than 30 seconds later.

This run used machines in two classes: older, slower MIPS based machines and relatively newer and faster machines with Sparc processors. On our application, the Sparcs are a little more than two times faster than the MIPS machines. Unfortunately, we have fewer Sparc machines, and the demand for them is much greater than for the MIPS machines. To utilize as many Sparc machines as it can get, the application removes a MIPS machine to allow itself to request a new Sparc machine when a software imposed upper limit of sixty machines is reached. If a Sparc machine is reclaimed by an owner, it will be replaced by the first available machine in either class. Because MIPS machines are plentiful in our pool, the application maintains nearly the maximum number of machines throughout the run. Figure 5 shows the result of this scheme. Initially, the application is able to rapidly acquire MIPS machines until the limit of sixty total machines is reached. At this point, it begins to remove MIPS machines in favor of Sparcs which are acquired much more slowly. Starting a little before hour eight, a number of Sparc machines become available, and they are used to replace MIPS machines.

5 Conclusions and Future Work

Initial experience with implementations based on our framework for separating resource management services from PPEs have been very encouraging. We have been able to modify an existing PPE to use our scheme with little difficulty. More significantly, we have been able to use the modified PPE to implement two separate RM schemes. By emulating

TABLE 1

Elapsed Time	Total Occupancy	Accumulated CPU Time	Spawn Occupancy	Suspend Time	Total Hosts
9:28:59	528:40:32	500:43:35	6:24:59	15:03:03	161

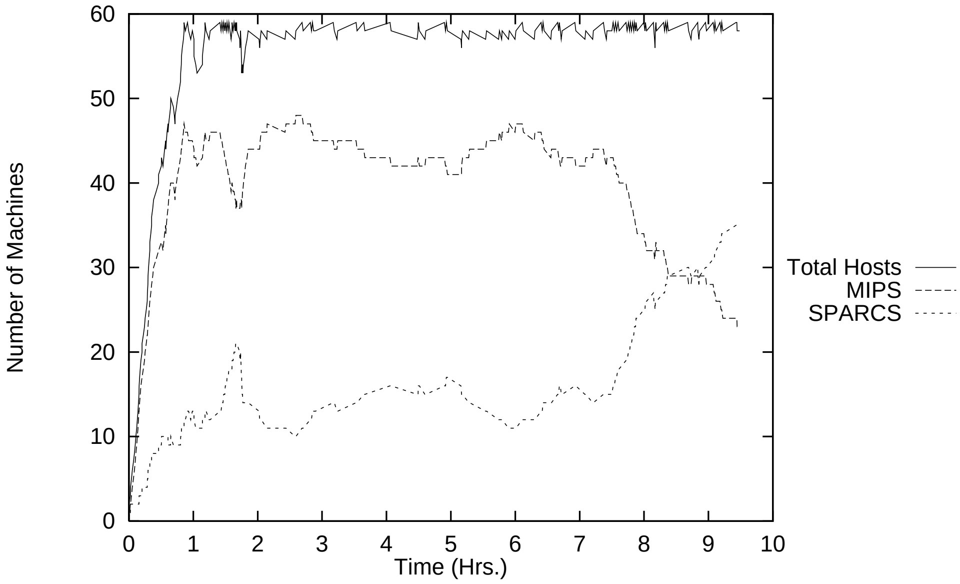

FIG. 5. *Host Occupancy*

the current handling of RM requests, we have shown that our scheme need not alter the behavior of PPEs users have become accustomed to. However, using Condor as a resource manager shows that the scheme is flexible enough to allow customization and extension to new RM services which can be highly beneficial to application writers.

In the future, we hope to work more with application writers to see how they are able to use the facilities provided by the Condor RM in their programs, and to see what new RM primitives may be useful. We also want to apply our architecture to new PPEs. A prime candidate is MPI because it is standardized, and because it does not specify any RM services. This allows us to use the experience gained with PVM to define a complete suite of resource management services which are useful to application writers.

References

[1] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor: A Hunter of Idle Workstations," *Proceedings of the 8th International Conference of Distributed Computing Systems,* San Jose, California, pp. 104-111, (June 13-17, 1988).

[2] D. Duke, T. Green, and J. Pasko, *Research Toward a Heterogeneous Networked Computing Cluster: The Distributed Queuing System Version 3.0,* Supercomputer Computations Research Institute, Florida State University, March 1994.

[3] A. Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3 User's Guide and Reference Manual.* Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.