Portland State University

PDXScholar

Computer Science Faculty Publications and Presentations

Computer Science

1-1995

Scheduling of Parallel Jobs on Dynamic, Heterogenous Networks

Dan Clark
Oregon Graduate Institute of Science & Technology

Jeremy Casas
Oregon Graduate Institute of Science & Technology

Steve Otto
Oregon Graduate Institute of Science & Technology

Robert Prouty
Oregon Graduate Institute of Science & Technology

Jonathan Walpole
Oregon Graduate Institute of Science & Technology

Follow this and additional works at: https://pdxscholar.library.pdx.edu/compsci_fac

 Part of the Computer and Systems Architecture Commons, and the OS and Networks Commons

Let us know how access to this document benefits you.

Citation Details

"Scheduling of Parallel Jobs on Dynamic, Heterogenous Networks," Dan L. Clark, Jeremy Casas, Steve W. Otto, Robert M. Prouty, Jonathan Walpole Technical report, Dept. of Comp. Sci., Oregon Graduate Institute of Science and Technology, January, 1995.

This Technical Report is brought to you for free and open access. It has been accepted for inclusion in Computer Science Faculty Publications and Presentations by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

Scheduling of Parallel Jobs on Dynamic, Heterogenous Networks

Dan L. Clark, Jeremy Casas, Steve W. Otto, Robert M. Prouty, Jonathan Walpole

{dclark, casas, otto, prouty, walpole}@cse.ogi.edu
http://www.cse.ogi.edu/DISC/projects/cpe/

January 31, 1995

Abstract

In using a shared network of workstations for parallel processing, it is not only important to consider heterogeneity and differences in processing power between the workstations but also the dynamics of the system as a whole. In such a computing environment where the use of resources vary as other applications consume and release resources, intelligent scheduling of the parallel jobs onto the available resources is essential to maximize resource utilization. Despite this realization, however, there are few systems available that provide an infrastructure for the easy development and testing of these intelligent schedulers. In this paper, an infrastructure is presented together with a scheduler that is capable of both gang scheduling and dynamic task reallocation of PVM applications.

1 Introduction

In recent years, the emergence of high-powered workstations connected via fast communication networks has increasingly been considered as an alternative to dedicated high-performance parallel computers. These workstation networks are not only cheaper, but also provide a general-purpose computing environment that is typically shared by both parallel and non-parallel application developers and users.

Software systems such as the Parallel Virtual Machine (PVM) [1, 2, 3] provide an infrastructure where such networks of workstations could be viewed by an applications developer as a large distributed-memory multi-processor machine. These software systems make it convenient for developers to create parallel applications by virtualizing the workstation network and by providing the necessary primitives for process control and communication.

Despite the success of these systems, there are areas such as resource allocation where these systems lack support. In a computing environment where the availability of resources change over time, the intelli-

gent allocation and re-allocation of resources in response to these changes is essential to utilize the resources effectively.

In this paper, an infrastructure for developing and testing intelligent schedulers is defined. A prototype implementation of the infrastructure based on the Migratable PVM (MPVM) [4] framework is presented together with a scheduler that is capable of deciding on task to workstation allocations based on workstation availability, non-uniform processing power, gang scheduling of multiple parallel jobs, and dynamic re-allocation of tasks as resource utilization varies. The final sections present related work and conclusions.

2 Conceptual Model

The conceptual model describes the functions, responsibilities and relationships between the abstract components of the infrastructure: the scheduler, load monitor, message passing system, task control facility and parallel applications. Organizing the infrastructure into these functional components provides a method to analyze the individual components and an opportunity for their concurrent design utilizing different areas of expertise. Based on this approach, the individual components are described below.

To facilitate this discussion, we will use the following terminology. The machines currently designated as part of the virtual machine are said to be *participating* and are referred to as *nodes*. A participating machine that is not currently being used by its owner is said to be *available*, while a machine which is being used by its owner is *unavailable*. Which *state* (available or unavailable) a machine is in is called its *availability.* The basic unit of scheduling is a *task*.

A load monitor (LM) runs on each participating machine. Each LM gathers load statistics on its host machine and sends this information to the scheduler. The most visible use of the load monitor is to allow for the automatic detection of owners reclaiming their workstation. The LM only contacts the scheduler when the availability of the host node or some other important load statistic changes significantly. How node availability is decided, or what constitutes a “significant change” in some other statistic is a policy decision and will not be discussed here. We would like to note that it is generally desirable for the LM to use as few system resources as possible, but there may be a trade-off between this goal and information accuracy.

The scheduler decides when and where all jobs run. In doing so, it must partition the resources among the competing jobs. Partitioning is performed in both space and time. Space partitioning is accomplished by placing jobs on non-overlapping portions of the network domain. Time partitioning is accomplished by

gang scheduling among the jobs. The scheduler is responsible for maintaining enough information about the state of both the nodes and the tasks on the network to make good decisions about job placement. The scheduler gets availability information from the LM. It responds to four basic events: (1) owner reclaiming a workstation, (2) owner leaving a workstation, (3) a new job entering the system, and (4) a job finishing. When any of these events occur, the scheduler reevaluates the current schedule, moves any tasks it needs to, and creates any new tasks.

The message passing layer supplies location transparent communications. This mechanism is used by the scheduler to communicate with the load monitor, the task control facility, and the application tasks. It is also the mechanism by which the various parts of the parallel jobs communicate. Figure 1 shows the interfaces among the infrastructure components.

The task control facility (TCF) provides an instantiation of the abstract unit of work for the scheduler. It is these tasks which are seen and manipulated by the scheduler. The TCF utilizes the services provided by the underlying OS to facilitate the creation, deletion, migration, suspension and resumption of individual tasks.

Application programs are written in terms of a variable, abstract number of tasks. In this way the program specifies the parallelism for a job, but does not know where a job will run nor how many machines will be used. Tasks access scheduling services (e.g. task creation and deletion) and communicate to each other through the message passing layer. Tasks can also choose to register handles for the notification of system level events such as migration, suspension, resumption and task exit.

3 Architecture and Interfaces

In this section, a prototype infrastructure for the development of intelligent schedulers is presented. MPVM, an extension of PVM 3.3.4 that provides transparent task migration, is used as the basis for the prototype implementation. MPVM was chosen for three reasons. First, MPVM already provides the functionality required for the task control facility and the message passing system. Task control in MPVM includes the ability to spawn, migrate, suspend, and resume execution of tasks. For the message passing system, MPVM provides message passing functionality such as automatic data translation and unicast/multicast messaging.

Second, MPVM allows a task to register itself as a resource manager or scheduler. Once registered as the scheduler, all application requests that require scheduling decisions (e.g., spawn) and system state

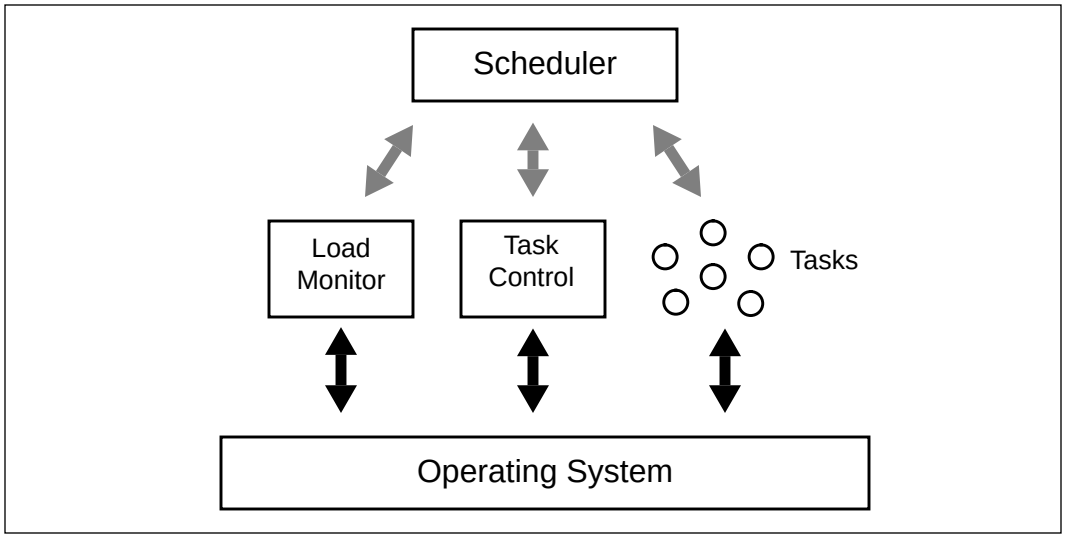

FIGURE 1. This figure illustrates the interfaces among the infrastructure components. Grey lines represent communication via the message passing interface. Black arrows represent some operating system specific services..

information (e.g., node becomes unavailable) are sent to the scheduler. This allows the scheduler to take control of all necessary scheduling decisions.

Lastly, MPVM is capable of asynchronously migrating tasks between homogeneous nodes. This capability allows for the development and testing of scheduling algorithms that use dynamic resource allocation. The migration mechanism is triggered by a message from the scheduler. The migration message specifies which task to migrate and the target node for migration. To asynchronously migrate tasks, a migrate signal is sent to the migrating task and causes a migration signal handler within the task to be invoked. The migration signal handler then saves the state of the task which includes the processor register values and the contents of its data/stack space. In the mean time, a new task is started on the target node using the same executable file from which the migrating task was started. Finally, the state information saved by the migrating task is transferred, through a TCP socket, to the new task. The new task then continues execution based on the received state. In addition to automating the migration of task state, MPVM also provides task id virtualization and message forwarding/sequencing to ensure that messages are not lost and are received in the correct order [5].

Just like standard PVM, MPVM requires a daemon to be running on each node in the system and applications to be linked with the MPVM message passing library. Note that except for the ability to migrate tasks and to asynchronously suspend/resume execution of tasks (without losing tty connections), all the other functionality was provided by standard PVM from which MPVM was based.

The other components of the conceptual model that have yet to be defined are the load monitors and the parallel applications. The parallel applications used are PVM programs with variable levels of parallelism. Such applications serve as good test cases for scheduling algorithms. Lastly, in this implementation, the load monitors are MPVM tasks which are executed on every node in the system, each sending load information to the scheduler about the nodes.

Figure 2 shows the organization of this infrastructure. The interface between the various components, in the form of messages, is shown in table 1.

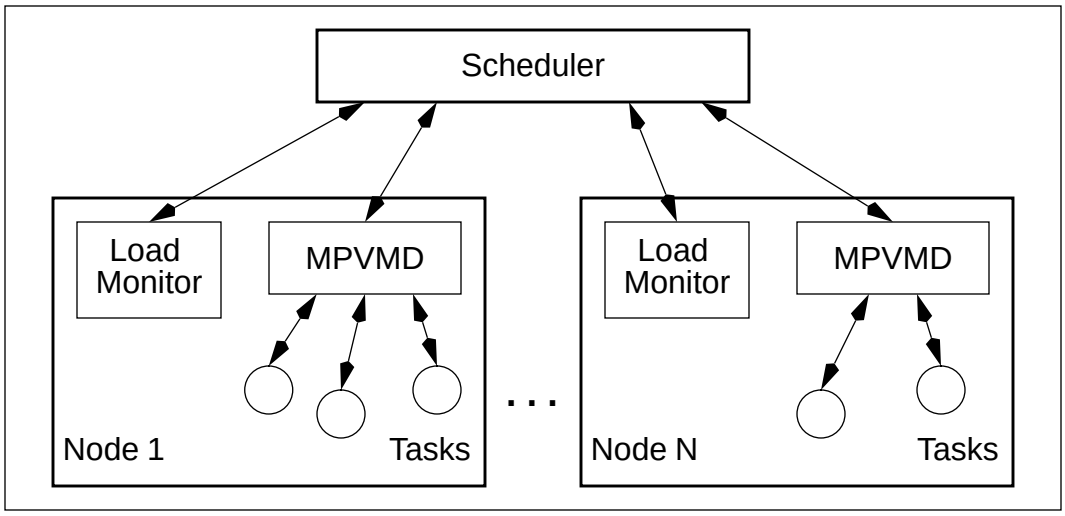

FIGURE 2. Organization of prototype infrastructure using MPVM. MPVM in this prototype implements both the task control facility and the message passing library. The tasks represent the parts of the parallel application.

The scheduler in this prototype maintains information about all the nodes in the system. This informa-

Message Type	Direction	
SM_SPAWN	T $\leftrightarrow$ S	task request to spawn tasks
SM_EXEC	S $\rightarrow$ D	scheduler directive to spawn tasks on specific nodes
SM_EXECACK	S $\leftarrow$ D	SM_EXEC acknowledgment from daemon
SM_TASKX	S $\leftarrow$ D	task exit notification
SM_MIG	S $\rightarrow$ D	scheduler directive to migrate tasks to another node
SM_MIGACK	S $\leftarrow$ D	SM_MIG acknowledgment from daemon
SM_SUSPEND	S $\rightarrow$ D	scheduler directive to suspend execution of a task
SM_RESUME	S $\rightarrow$ D	scheduler directive to resume execution of a task
SM_KILL	S $\rightarrow$ D	scheduler directive to terminate a task
SM_LOAD	S $\leftarrow$ L	status information from load monitor
S = Scheduler, T = Task, D = MPVM Daemon, and L = Load Monitor		

TABLE 1. Message interface between the different components of the prototype infrastructure

tion includes the node's name, processor type, processing power, availability (from the load monitors), etc. This collection of information about the nodes in the system, if used, could result in more intelligent scheduling decisions. In addition to the node information, the scheduler also maintains information about all the tasks in the system, and the task-to-node mappings in particular.

The scheduler also provides an X Windows graphical user interface (GUI) which can be used to visualize the contents of key data structures used by the scheduler. These data structures include the names, identification numbers, and availability of the nodes used, and the identification number, current location, and job identifier of tasks in the system. The GUI does timely updates of the information on screen as the state of the system changes (e.g., a task migrates). The GUI also provides a simple drag-and-drop interface for manually rescheduling tasks in the system and a push button interface for changing the availability of individual nodes. An important point to note about the scheduler is that it is able to trap messages that require scheduling decisions. Spawn request messages, load information messages, task termination messages are all examples of such messages. By trapping these messages, the scheduler can use its own policy to make resource allocation decisions to achieve better system utilization.

The load monitors in the prototype infrastructure, aside from being MPVM tasks themselves, have X Windows GUIs which present a single push button interface. This allows a user to manually toggle the availability of their workstation by pressing the button. When pressed, the load monitor sends the scheduler a message containing the current availability of the workstation.

Figure 3 illustrates the interaction between the various components as the status of a node changes. In this example, the load monitor is toggled by the user on node 1 changing the availability to unavailable. This change in availability prompts the load monitor to send an SM_LOAD message containing the status of "unavailable" to the scheduler. The scheduler, noting that T_1 is executing on node 1, determines a new schedule and sends an SM_MIG message to the daemon on that node to migrate task T_1 to node 2. The daemon then migrates the task to node 2. The migration sequence is shown in light-colored arrows in the figure since this sequence of events is internal to MPVM and is of no concern to the scheduler. Finally, the migrated task T_1' resumes execution and the daemon on node 2 sends an SM_MIGACK back to the scheduler acknowledging that the task has successfully migrated and is again executing.

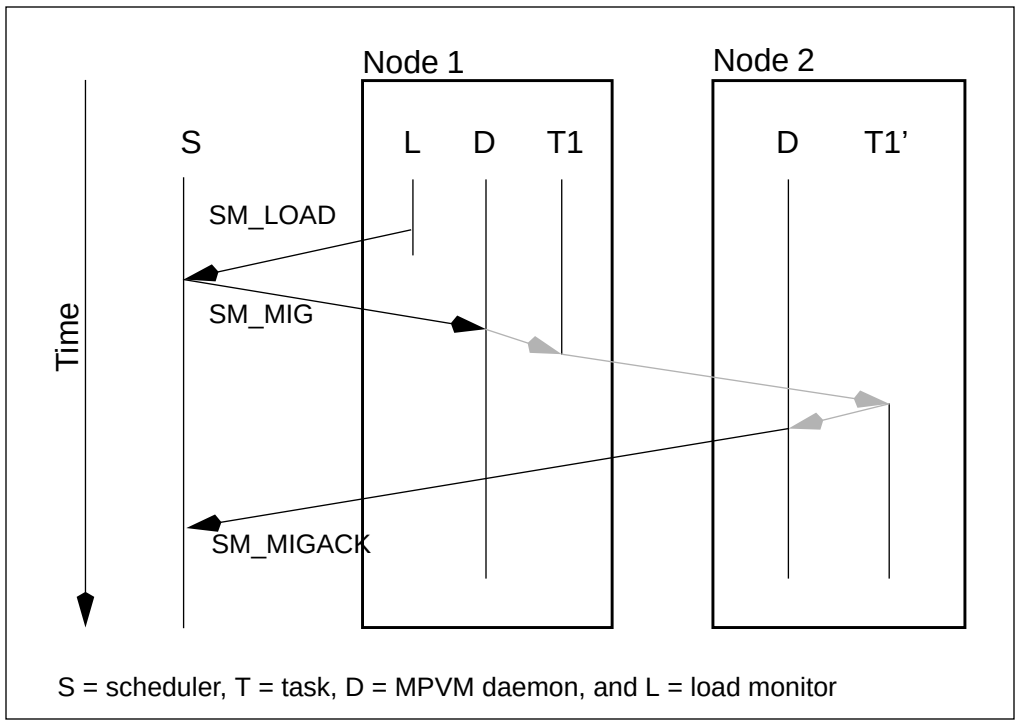

FIGURE 3. Example interaction between the various components as the status of a node changes. The light-shaded arrows interaction internal to MPVM.

4 Summary of Scheduling Policy

The guiding principles behind the scheduler are providing fair access and better utilization of the nodes on a heterogenous network of computers. In order to provide fair access, it is necessary to define quantifiable measures for the capabilities of the node resources and a method to divide the resources between com-

peting jobs. The measured quantities should be available across all the nodes, easy to determine, and obtainable unobtrusively. Two different methods of partitioning the resources are by dividing the nodes among tasks or time slicing among tasks.

Distributing the tasks to the idle nodes will utilize the idle cycles on the network more fully. This distribution of the tasks should not jeopardize the overall job response time. Also, the tasks of an individual job should be scheduled to run simultaneously on all the nodes allocated to the job

One measure of fair access is to ensure that each job gets access to at least as much CPU power available from a network with no jobs as it does from some smaller piece. For a small number tasks compared to nodes the jobs can be divided up across the nodes in the network. With a sufficient demand it will become necessary to share individual nodes among multiple jobs.

The framework of the scheduler can be used to test many different scheduling policies. The first implementation uses gang scheduling of parallel jobs to verify performance advantages for a distributed memory architecture. In order to provide fair progress to all jobs, the scheduler time slices among jobs competing for a node, shown in figure 3. In order to make maximum use of the available CPU capacity in a time-slice, the jobs are packed together by placing a job within the free space in a time slice. The jobs are only packed together when the processing capability of the free space is equal to placing a job in a new time slice.

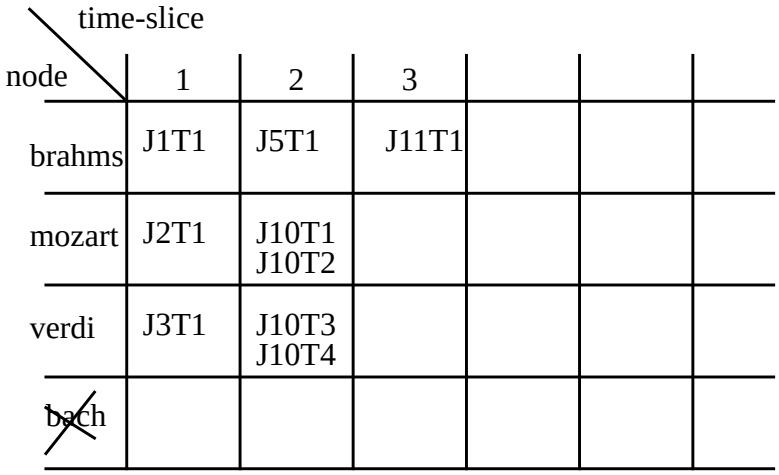

FIGURE 4. The gang scheduling of tasks on a network partitioned by time slices and node domains. The node bach is unavailable for scheduling. Time slice 3 has available capacity on nodes mozart and verdi.

The determination of the mapping between the tasks and the nodes on the network is accomplished by implementing the scheduling algorithm of Al-Saqabi [6]. The algorithm takes three basic steps to mathematically determine the allocation of tasks to nodes: distribute, compress and pack. The first step determines hypothetically how fast the job will run given its own time slice distributing the tasks proportionally across all the nodes. The second step attempts to find a smaller set of nodes that will run the job just as fast. The third step then looks at a subset of the possible combinations of free space available among all the jobs to pack the job into the existing schedule to avoid the unnecessary creation of a new time slice.

The mathematical algorithm used in the determination of the distribute step uses the node CPU speed available from the MPVM daemons. The algorithm normalizes the CPU speed across all the nodes by the slowest speed to determine the relative speed of each node. The component of the job allocated to each node is then determined by multiplying the total request of the job by the relative node speeds and rounding down to an integer value. The fraction of the job not allocated to any job is allocated to the node with the maximum available processing capability left over with a bias towards nodes which already have tasks running on them. The computation of this step of the algorithm is bounded by the sort of the nodes by the relative speed. The result of the distribute step determines one of many potential mappings of whole tasks to nodes. This gives a hypothetical optimal performance of the job assuming equal work of all the component tasks by spreading the job among all the workstations.

The second step attempts to pack the tasks onto fewer nodes while still yielding optimal performance. The effect of this step is to uniformly double up the tasks on fewer nodes, for example J10 in figure 3 illustrates a packing of a four tasks of a single job onto two nodes. This is accomplished by the following steps. First, calculate the maximum capacity of each node based on the mapping which occurred in the first step. Second, eliminate the nodes which do not yield an integer number of task assignments. Finally, if there is excess capacity, map the tasks to the nodes attempting to free up the maximum number of nodes. The advantage of freeing up nodes is that the free nodes in this time slice can be used to schedule the tasks of other jobs.

The last step is to determine if the job can be packed into any available free space such that the new schedule will yield equally optimal performance. In figure 3 the three jobs J1, J2, and J3 have been packed into the first time slice. This is achieved by a partial search through the time slices currently active on the system to determine if there is any available CPU cycles in the schedule which will match the ideal performance resulting from a new time slice.

The algorithm described above is used both for initial task assignment on nodes and for rescheduling of tasks based on dynamic changes to the jobs or the nodes. The view of the network used by the algorithm is constrained by the homogenous boundaries imposed by process migration and the need to fix some tasks to nodes but does not change the mathematical foundation of the task placement.

5 Related Work

The Parallel Virtual Machine defines an interface for specifying a resource manager, conveniently separating scheduling mechanism from policy [7]. The interface clears the way for the development and testing of different scheduling algorithms for use with the PVM system, and is used in our prototype infrastructure. As a resource manager, it receives service requests/information that could affect scheduling decisions like spawn requests or task termination. In using this interface for our prototype infrastructure/ scheduler, the interface was extended to included messages regarding load information and support for task migration which was available through MPVM.

The work at the University of Wisconsin-Madison by Livny [8, 9] integrating the scheduling mechanisms of Condor with PVM provided the impetus for the creation of a well defined interface for resource management which is now part of the standard release of PVM. Condor supports process migration of sequential jobs. Currently, Condor's supports for PVM applications is only up to task execution suspension/resumption. In our infrastructure, we have extended this capability, with the use of MPVM, to support migration of tasks.

Neuman specifies the a framework for scalable resource management on distributed system using a system manager, job manager, and a node manager for the Prospero system [10]. The system manager is responsible for allocation of resources among multiple jobs leaving the job manager to assign the specific tasks of a job to a node. The allocation of system wide resources is combined into a single function under in our infrastructure. The node manager in Prospero is responsible for task management, communication and load monitoring which is divided between the MPVM daemon and a new load manager function in our system.

The scheduling mechanism of the MOSIX [11] operating system utilizes load values and normalized CPU capacity to determine the location for task placement in a distributed system. The system was implemented by modifying the Berkeley 4.3 kernel which allows the system to provide the normal UNIX interface for task invocation. Utilizing the active monitoring of processor load and transparent process

migration the tasks can be dynamically relocated to under utilized nodes on the network. In our infrastructure, all the mechanisms were implemented at user-level, requiring no kernel modifications. By not building upon mechanisms tied to specific operating systems or even hardware, in general, our infrastructure can be used on a larger set of systems/machines.

The Distributed Queueing System (DQS) from Florida State University provides support for batch scheduling of multiple jobs across a heterogenous network [12]. DQS also supports parallel jobs under PVM by starting up a new set of PVM daemons at the beginning of a job and shutting down the PVM daemons at job termination adding the overhead of the PVM system initiation to the cost of running the job. The DQS system is similar to our work because it uses a central queue master to determine the distribution of work to all the remote system and limits the number of tasks running on a node to only the tasks of a single job. Our work allows for dynamic reallocation of tasks and by providing progress to all jobs through the use of time slicing between competing jobs.

Aside from the systems mentioned above, there are many other scheduling related software systems available such batch schedulers (LSF, NQS, etc.), schedulers available from commercial multi-processor vendors (Intel Paragon, IBM SP2, etc.), and distributed operating systems (Sprite, Amoeba, etc.).

6 Conclusions and Future Work

The conceptual framework with the abstract components of a scheduler, a task control system, a load monitor and tasks is used to define the architecture for the development of intelligent schedulers for parallel jobs. The implementation of the components within the MPVM system effectively illustrates the integration of the new migration mechanisms with a dynamic scheduling policy. The PVM 3.3 system serves as a basis for the implementation but needs to be extended to have message support for process migration and load information for the nodes. The addition of the load monitoring facilities provides the scheduler with a more dynamic view of the nodes on the network.

The use of a separate scheduler for tracking and placing the tasks on the network provides a facility for implementing new scheduling policies for managing parallel job tasks. The first algorithm which was implemented uses relative CPU speeds to determine the allocation of tasks to nodes to provide use of the resource proportional to their capabilities. The scheduler also uses round robin time slicing to provide fair access to the node resources for competing jobs.

In the future we would like to implement alternative scheduling schemes and compare performance results. The definition of a clear set of metrics for evaluating performance also needs to be determined. These metrics can then be used to run an appropriate set of benchmark applications to evaluate the quality of the scheduling framework and algorithm. Another area of interest is in the development of distributed scheduler within our infrastructure.

7 References

[1] A. L. Beguelin, J. J. Dongarra, A. Geist, and R. J. M. V. S. Sunderam. "Heterogeneous network computing." In *Sixth SIAM Conference on Parallel Processing*. SIAM, 1993.

[2] J. J. Dongarra, A. Geist, R. J. Manchek, and V. S. Sunderam. "Integrated PVM framework supports heterogeneous network computing." *Computers in Physics*, April 1993.

[3] A. L. Beguelin, J. J. Dongarra, A. Geist, R. J. Manchek, S. W. Otto, and J. Walpole. "PVM: Experiences, current status and future direction." In *Supercomputing '93 Proceedings*, pages 765–6, 1993.

[4] J. Casas, R. Konuru, S. Otto, R. Prouty, and J. Walpole. "Adaptive load migration systems for PVM." In *Supercomputing '94 Proceedings*, pages 390–399, Washington D.C, November 14-18 1994.

[5] J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walpole. "MPVM: A migration transparent version of PVM." Technical report, Oregon Graduate Institute of Science & Technology, 1995.

[6] K. A. Saqabi, S. W. Otto, and J. Walpole. "Gang scheduling in heterogenous distributed systems." Technical report, Oregon Graduate Institute of Science & Technology, 1994.

[7] R. J. Manchek, "PVM version 3.3.0 release-notes," June 1994. Anonymous ftp site netlib2.cs.utk.edu directory /pvm3.

[8] M. J. Litzkow, M. Livny, and M. W. Mutka. "Condor – A hunter of idle workstations." In *Proceedings of the 8th IEEE International Conference on Distributed Computing Systems*, pages 104–111, June 1988.

[9] J. Pruyne and M. Livny. "Providing resource management services to parallel applications." In *Proceeding of the 2nd workshop on Environments and Tools for Parallel Scientific Computing*, pages 152–161, 1995.

[10] B. C. Neuman and S. Rao. "The prospero resource manager: A scalable framework for processor allocation in distributed systems." In *Concurrency: Practice and Experience*, volume 6(4), pages 339–355. Wiley, June 1994.

[11] A. Barak, S. Guday, and R. G. Wheeler. *The MOSIX Distributed Operating System – Load Balancing for Unix*. Lecture Notes in Computer Science. Springer-Verlag, 1993.

[12] T. Green, R. Pennington, and D. Reynolds. *Distributed Queueing System Version 2.1 Release Notes*, March 1993. Release Notes.