

# MIST: PVM with Transparent Migration and Checkpointing

Jeremy Casas, Dan Clark, Phil Galbiati, Ravi Konuru,  
Steve Otto, Robert Prouty and Jonathan Walpole  
mist@cse.ogi.edu  
<http://www.cse.ogi.edu/DISC/projects/mist/>

Department of Computer Science and Engineering  
Oregon Graduate Institute of Science & Technology  
PO Box 91000 Portland, OR 97291-1000, USA

May 1995

## Abstract

*We are currently involved in research to enable PVM to take advantage of shared networks of workstations (NOWs) more effectively. In such a computing environment, it is important to utilize workstations unobtrusively and recover from machine failures. Towards this goal, we have enhanced PVM with transparent task migration, checkpointing, and global scheduling. These enhancements are part of the MIST project which takes an open systems approach in developing a cohesive, distributed parallel computing environment. This open systems approach promotes plug-and-play integration of independently developed modules, such as Condor, DQS, AVS, Prospero, XPVM, PIOUS, Ptools, etc.*

*Transparent task migration, in conjunction with a global scheduler, facilitates the use of shared NOWs by allowing parallel jobs to unobtrusively utilize nodes that are currently unused. PVM tasks can be moved onto nodes that are otherwise idle, and moved off when the node is no longer free. Experiments show that migration performance is limited by the bandwidth of the underlying network. E.g. An 8 MB process migrates in 8 seconds on a 10 Mbps ethernet.*

*We have implemented a global scheduler as a PVM resource manager which can take advantage of task migration to perform dynamic scheduling of tasks. Some extensions to the resource manager interface were required. The task migration mechanism also serves as the basis for transparent checkpointing, which is a common method for improving a system's fault-tolerance. We have developed a PVM prototype that integrates checkpointing and migration. This paper*

*presents an overview of the entire system, issues raised by this work, and discusses future plans.*

## 1 Introduction

PVM [1, 2, 3] is a widely used, public-domain software system that allows a heterogeneous network of parallel and serial computers to be programmed as a single computational resource. This resource appears to the application programmer as a potentially large message-passing virtual computer. Systems like PVM allow the computing power of widely available, general-purpose computer networks to be harnessed for parallel processing.

As both PVM users/developers and PVM applications matured, it became evident that PVM requires more support for intelligent scheduling and resource management, file management, program tracing/debugging, etc. The Migration and Integrated Scheduling Tools (MIST) project is working on enhancing PVM to address these needs. First, the project plans to provide support for intelligent resource management for PVM applications. Intelligent resource management includes multi-user support and unobtrusive access to idle cycles available on shared networks. Secondly, the project aims to integrate tools for debugging, profiling, and monitoring of parallel applications. In addition, the project will also develop tools for monitoring overall system utilization, resource availability, and network traffic.

The MIST project vision is shown in figure 1. Instead of building each of the components ourselves, we adopt an open systems approach wherein interfaces between the various components are well de-

fined. These interfaces would allow the various components to be used independently, and would allow interoperability in related environments. This open systems approach promotes “plug-and-play” integration of different, independently developed components.

In the context of this vision, this paper presents the mechanisms we have for supporting intelligent scheduling and resource management. We present the prototypes we have for the MIST kernel, the scheduler, and the system load monitor components. For the MIST kernel, we use MMPVM [4] (Multi-user, Migratable PVM), an enhanced version of PVM that supports transparent task migration, application checkpointing, and multi-user application execution. For the scheduler, we make use of an enhanced version of resource manager interface provided by PVM. The enhancements to the resource manager interface were made to enable the scheduler to use the new features of MMPVM. Lastly, for system load monitoring, we use a simple interface that allows users to toggle the availability/unavailability of individual machines.

The remainder of the paper is organized as follows. Section 2 presents an overview on the need for and implementation of intelligent schedulers. The implementation of MMPVM, the scheduler, and the load monitor is presented in section 3. A general discussion regarding our experience with the prototype MIST system and various issues presented by this work are presented in section 4. Related and future work are discussed in sections 5 and 6. A summary is then provided in section 7.

## 2 Overview

PVM is typically used in a computing environment composed of a shared network of workstations. In using such a computing environment for parallel processing, it is important to recognize that machine utilization is generally unpredictable. This unpredictability can be caused by other users running jobs of varying computing requirements. It could also be caused by machine owners as they allow/disallow usage of their machines. The implication of this unpredictability is that the performance of PVM applications could be severely penalized as one or more of the machines the PVM application is using becomes heavily used. It becomes worse when owners want to reclaim their machines, forcing other users to remove their applications. Removal of applications from machines often result in the application being terminated.

Another important part of this unpredictability is the possibility of workstations being shutdown or sim-

ply failing. In this case, it is most likely that the failed application will have to be restarted from the beginning, losing all the results it has already computed.

For these reasons, we have added task migration and checkpoint/restart functionality into PVM. Task migration allows tasks to move from one workstation to another as the workstation they are on becomes heavily used or is reclaimed by its owner. Checkpoint/restart on the other hand guarantees application progress in the presence of failures by allowing an application to restart in a state other than at the very beginning.

While task migration and checkpoint/restart functionality are necessary, they are not sufficient to be able to effectively utilize the workstations in a network. It is also necessary to have a scheduler that can determine which tasks should run on which machines to give the best possible overall performance (or an approximate thereof) without being obtrusive to machine owners. The scheduler would be responsible for deciding a) the initial placement of application tasks, b) when and where a task should migrate, and c) when to checkpoint/restart applications. PVM (as of version 3.3.2) provides a basic interface to such external schedulers or resource managers. We had to extend this interface to accommodate task migration and checkpoint/restart.

A scheduler, however, can only make decisions based on what it knows. At the very least, a scheduler would require knowledge of what tasks to schedule and on what machines it could schedule them on. The first requirement of knowing which tasks to schedule is addressed by providing a multi-user version of PVM. Traditionally, PVM is run on a per user basis. This per-user property has good qualities such as security and isolation (a bug in one user’s PVM application will not affect those of other users). For a scheduler to make good “global” scheduling decisions, however, it is necessary for the scheduler to know of all PVM tasks on the system regardless of the user. Making PVM multi-user gives the scheduler complete knowledge about all PVM tasks running on the system.

The second requirement of knowing which machines are available for scheduling is addressed by the use of load monitors. Load monitors provide the scheduler with information such as the current system load/utilization or whether the machine was reclaimed/vacated by its owner. This information allows the scheduler to determine which machines can and cannot be used.

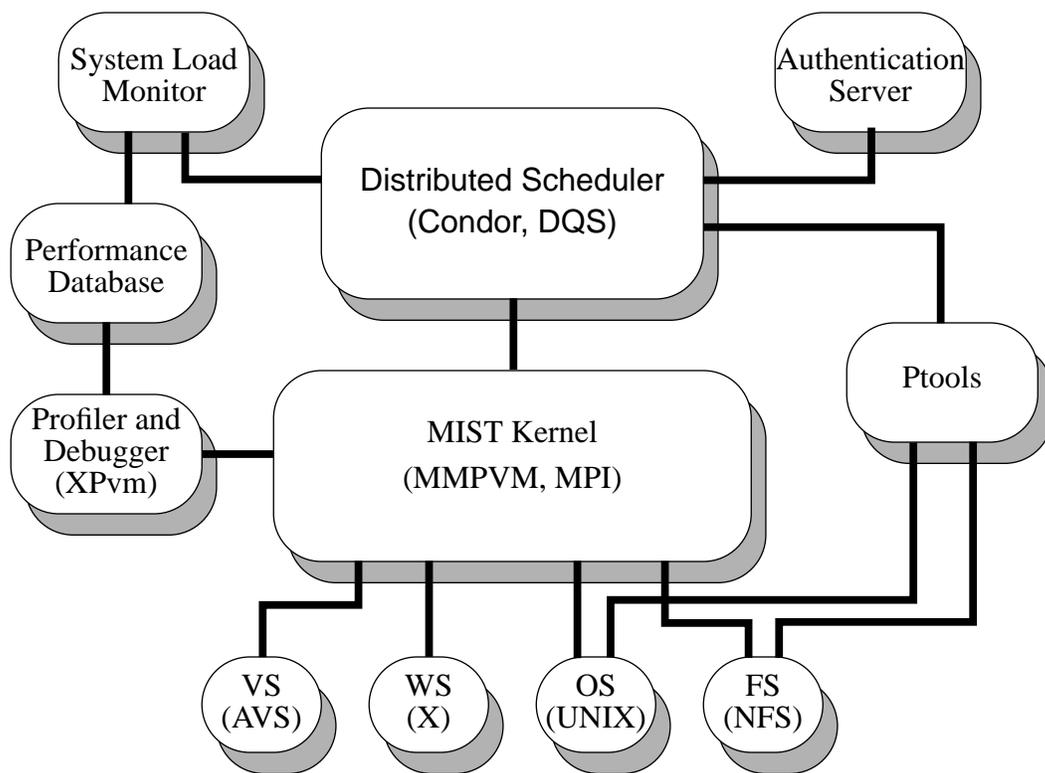


Figure 1: MIST Project Vision.

### 3 Implementation

In this section, we present a high-level description of the implementation of MMPVM, the scheduler, and the load monitors. The prototype described in this section has been around for quite some time. We’ve been using it for weeks at a time for executing long running parallel search applications for solving combinatorial problems such as the Traveling Sales Person problems. Within this time, we’ve seen tasks move around as workstation owners allow/disallow usage of their workstations. While we have yet to quantify the performance of the MIST system and its effect on the applications, it is certainly up and running.

#### 3.1 MMPVM

MMPVM [4] is an enhanced version of PVM that is capable of transparent process migration, application checkpointing/restart, and running applications of different users. It is implemented entirely at user-level, using facilities available through standard Unix system calls and libraries. The advantage of such an implementation is that there is no need for kernel modification, making it portable to various Unix flavors.

Process migration is the ability to suspend the execution of a process on one machine and subsequently resume its execution on another. This functionality requires the ability to save the state of a process on one machine and restore it on another.

The state of a PVM task can be viewed in two ways: its state as a Unix process and its state in relation to the PVM application. As a Unix process, its state includes a) the processor state (e.g., machine register values), b) the state held by the process itself (e.g., text, data, stack) and c) the state held by the OS for the process (e.g., state of open files). As part of a PVM application, the state of a task includes its task ID and the messages sent from/to that task. For PVM task migration to be realized, all state information should be captured, transferred, and re-constructed.

To maximize migration performance. MMPVM supports asynchronous task migration. Asynchronous task migration allows a task to migrate independent of the what the other tasks in the application are doing. Fundamental to correct PVM application behavior in the presence of asynchronous migration are the concepts of task ID virtualization, message forwarding, and message sequencing. Task ID virtualization allows tasks to communicate with each other regard-

less of their real location. In PVM, the host ID of the machine a task is on is encoded in its task ID [5]. In MMPVM, the host number encoded in a task ID may no longer be that of the host the task is really on. Task ID virtualization is achieved by maintaining task-to-host mappings on the pvmds. These task-to-host mappings are updated whenever tasks migrate. Message forwarding, on the other hand, is necessary to ensure delivery of messages. A consequence of task migration is that messages may be sent to the wrong host (e.g., a message is sent to a host and the target task just migrated). In this case, the message would have to be forwarded to the correct location of the task. A side-effect of message forwarding is that multiple messages for a task may take different routes, causing these messages to arrive at the target task in a different order than when they were sent. This situation is in direct violation of PVM message passing semantics. By using a message sequencing mechanism, messages are guaranteed to be received by a task in the same sequence as they were sent.

Table 1 shows the migration performance as measured between two HP 9000/720 workstations running HP-UX 9.03 connected over an idle 10 Mb/sec Ethernet. The application used was a parallel Gaussian elimination program.

The process state size indicates the amount of data transferred as measured at migration time. The obtrusiveness cost indicates the amount of time from when the task was told to migrate to when it vacated the machine. The migration cost is the amount of time from when the task was told to migrate to when it restarted on another machine. Lastly, the TCP transfer time is the actual transfer time of just the process state through the network. This measure gives us a lower bound on achievable migration speed.

While the use of a user-level implementation allows for ease of porting, it does prevent migration from being totally transparent. Process IDs, for example, cannot be guaranteed to be the same on the target host. However, there is other OS held state information that can be saved and restored. For example, information about open files can be captured by the migrating task and transferred along with the data. The skeleton process can then use this information to re-open the files and restore their state (e.g., file pointer offset).

A hard limitation that must be recognized is that in using this task migration implementation, tasks can only be migrated between homogeneous machines. The fundamental problem here is that translation of process state information as saved in one architecture to another is difficult.

Having implemented the migration mechanism, building the core checkpoint/restart mechanism was straight-forward. In migration, process state is sent by the migrating task through a socket. In checkpointing, this process state is instead stored on disk. On restart, a skeleton process is spawned just like in migration but reads its data and stack from disk rather than a socket. The additional benefit of using the same migration mechanisms for checkpoint/restart is that a task can be restarted on a host other than where it was checkpointed, just as if it were migrated.

The current implementation of application checkpointing is synchronous. Before an application can be checkpointed, all the tasks in the application must first stop executing and agree to checkpoint. While there are other methods such as message logging, we opted for this approach since 1) it is simple, 2) does not add any overhead on the normal case, and 3) we expect checkpoints to be infrequent. A similar synchronization appears on restart where all tasks must first agree that everyone has successfully restarted before execution can proceed.

While the synchronization required by the checkpoint mechanism is quite heavy, the major cost in checkpointing is the actual saving of process state to disk. To minimize the impact of checkpointing on the application, the parallel application is allowed to execute while its state is being saved. This is done by letting the tasks execute a `fork()` system call where the parent process continues executing while the child process saves its state to disk.

When checkpointing a PVM application, special consideration should be given to the group server. The problem is that in the PVM system, different PVM applications that use group functions use the same group server. The group server contains state information about different PVM applications and thus cannot be checkpointed and restarted like other tasks.

The solution is to make the group server save application specific data. When a particular PVM application is checkpointed, the group server is also informed of which application is being checkpointed. This information causes the group server to save group state information about a particular application. When an application is restarted, the group server is again notified, causing it to reload group state information saved in a previous checkpoint.

The current implementation of our multi-user MMPVM requires that the pvmds run as root. This requirement is brought about by the need to change the user and group IDs of a spawned, migrated, or restarted task. The advantage of this approach is that

Matrix size	Process state size (bytes)	Obtrusiveness cost (sec)	Migration cost (sec)	TCP transfer time (sec)
0x0	97448	0.139	0.327	0.092
80 x 80	109736	0.257	0.361	0.103
300 x 300	277672	0.363	0.590	0.255
500 x 500	597160	0.683	0.871	0.549
1000 x 1000	2100392	1.993	2.205	1.924
2000 x 2000	8109224	7.512	8.324	7.449

Table 1: Obtrusiveness and migration costs for various matrix sizes. The process state size indicates the actual number of bytes transferred at migration time while the TCP transfer time indicates the time spent in sending the appropriate amount of data through a TCP socket connection.

it is simple and only requires one pvmd on each host for all users. The disadvantage of this approach is that the pvmds must run as root and users are no longer isolated from each other (i.e., my application could affect another user’s application).

One last point to mention is that MMPVM is also capable of suspending and resuming execution of tasks. This capability is useful if one wants to build gang scheduling capability into the scheduler, for example. Execution state (suspended or resumed) of tasks is persistent across migrations. That is, a suspended task can be migrated and the resulting task in the target machine goes back to suspend mode.

### 3.2 Global Scheduler

The Global Scheduler (GS), also known as the Resource Manager in the standard PVM distribution, is responsible for adding/deleting hosts from the virtual machine, deciding on where to spawn new tasks, detecting when tasks have terminated, and responding to queries for virtual machine configuration and task listings. The GS communicates with both the pvmds and the tasks through a well defined interface. This interface is defined in terms of reserved message tags.

For example, if a GS is registered to PVM and a task calls `pvm_spawn()`, a message is sent to the GS instead of the local pvmd. This message is tagged with the `SM_SPAWN` reserved message tag which the GS uses to determine that the message is a spawn request. Similar tags exist for other PVM calls such as `pvm_addhosts()`, `pvm_config()`, `pvm_notify()`, etc.

Within the context of our work, this interface provided us with a convenient way to interface with the GS. We then extended this interface to allow the GS to make use of the new capabilities of MMPVM. The extensions are shown in Table 2.

Using this extended interface, we have an implementation of the scheduling algorithm of Al-Saqabi [6]. This scheduling algorithm takes into account processor heterogeneity in terms of architecture and speed and the availability/unavailability of workstations to schedule tasks in the virtual machine. It recognizes the difference between cooperating tasks (tasks of the same application) and competing tasks (tasks of different applications) and schedules them accordingly to minimize contention.

Using the extensions provided by MMPVM, the scheduler is capable of gang scheduling tasks of multiple applications and is able to move tasks around as machines become available/unavailable or as tasks are made to double-up on processors. Processor doubling is a technique that allows an application to use fewer resources and yet perform as if all available resources were allocated to it. The extensions for checkpoint/restart have yet to be integrated with the scheduler but we plan to do this soon.

The prototype scheduler comes with an X-windows interface that allows us to visualize the current mapping of tasks to processors. It also provides drag-and-drop capabilities to force the migration of a task to a host and also allows us to toggle the availability status of hosts. In addition to being a nice visual component to the scheduler, it is a useful debugging tool and it allows us to monitor the current state of the system.

### 3.3 Load Monitor

The Load Monitor (LM) provides machine status information to the scheduler. There is one LM executing on each host in the virtual machine. It determines when the status of its host machine changes and informs the scheduler accordingly. There are a variety of ways to obtain load information (mechanism) and how to interpret the information (policy) for determin-

Message Tag	Direction	Purpose
SM_SUSPEND	GS → D	suspend execution of tasks
SM_RESUME	GS → D	resume execution of tasks
SM_MIG	GS → D	migrate tasks
SM_MIGACK	GS ← D	SM_MIG acknowledgement
SM_CKPT	GS → D	checkpoint tasks
SM_CKPTACK	GS ← D	SM_CKPT acknowledgement
SM_RESTART	GS → D	restart tasks
SM_RESTARTACK	GS ← D	SM_RESTART acknowledgement
SM_LOAD_REG	GS ↔ LM	register load monitor
SM_LOAD	GS → LM	machine load information

GS = Global Scheduler, D = MMPVM daemon, LM = Load Monitor

Table 2: Global Scheduler interface extensions.

ing availability of hosts. Regardless of which mechanism/policy is used, it is generally desirable for the LM to use as few resources as possible to avoid contributing significantly to the system load. There is usually a resource usage vs. information accuracy trade-off, depending on implementation.

The interface between the LM and the GS consists of two reserved message tags also shown in Table 2. The **SM\_LOAD\_REG** message tag allows the LM to introduce itself to the GS. The same message tag is used in the reply message from the GS acknowledging the registration of the LM for the machine it is executing on. The acknowledgment message also contains the current status of the machine, from the GS’ point-of-view, to initialize the LM. The **SM\_LOAD** is then used by the LM to inform the GS about subsequent changes in machine load status.

The load monitor we currently use gives the users explicit control over any node’s status. The mechanism takes the form of a “toggle switch” with both an X-Windows and command-line interface with which users can explicitly set the status of a node.

## 4 Discussion

In this section, we present some issues raised by this work. Note that the issues mentioned in this section involve all the components in the MIST project, not just those prototyped and presented in this paper. We realize that some issues discussed here have already been addressed and solved by other work such as those presented in section 5. But since we have yet to fully address these issues in our work, we include them here. There are some issues to which we propose possible/partial solutions and some which we just state as a problem that must be addressed.

### 4.1 File I/O

The introduction of migration and checkpoint/restart functionality into PVM raises some issues regarding the accessibility of files from different hosts. Three kinds of files are of particular importance: application binaries, application data files, and checkpoint files.

**Application binaries.** In standard PVM, application binaries need not be available on all hosts since the user can control which task starts (and hence which binary to use) on which host. In this case, only the binary file of a specific task needs to be on a specific host.

However, part of the migration protocol is the execution of a skeleton process on the target host using the same binary file from which the migrating task was started. This requirement implies that the application binary should be accessible on the target host. A similar situation arises when restarting a task on a host other than where it was checkpointed. Clearly, the availability of binaries limit where tasks can be migrated or restarted.

Availability of all binaries on all hosts can be accomplished using a global file system. If a global file system is not available, an alternative is to explicitly provide each host a copy of the binaries before the application is started.

**Application data files.** Unlike application binaries, data files pose a real problem. If the data files are used primarily for input (read-only), then just like application binaries, each host could be given a copy. Output files have to be treated differently since they are generated at run-time. Correct and transparent migration of a task that is writing to an output file depends on the accessibility of the output file on the target host.

Again, this requirement is easily satisfied by a global file system. If a global file system does not exist however, then there should be a mechanism for redirecting file I/O operations of the task such that the correct output file is updated.

**Checkpoint files.** When an application is checkpointed, a number of checkpoint files are generated. Only hosts that have access to these checkpoint files can be used to restart the application. The use of a highly-available global file system would be ideal in this case. If a global file system is not available, however, then the checkpoint files need to be stored in a highly available site (i.e., a file server). Restartability of the application would depend only on the availability of the file server. If this dependency on a particular file server is still unacceptable, the checkpoint files could be replicated on other places to increase availability.

These three kinds of files present different file I/O requirements in the presence of migration and checkpoint/restart. While these requirements are easily satisfied using a global file system, such global file systems are not as common as we would like them to be.

In a PVM system that uses multiple file systems, the effect of having a global file system could be accomplished using NFS, for example. Note that there is no need for the entire file system to be exported. Only the directories that would contain the PVM application binaries and data need to be exported. The use of NFS in this way implies that a strict naming convention should be used so that files could be accessed using the same path name regardless of location. The advantage of this method is that once it is set-up file access will automatically be handled by NFS.

An alternative to emulating a global file system using NFS is to provide an equivalent to a distributed file system on top of PVM. That is, all file I/O operations done by PVM applications, possibly including those done by the pvmds, are filtered to give the effect of a global file system. While this is more difficult to implement, it has the advantage of not relying on any underlying network file system.

One final comment on file I/O is with regards to idempotent file operations. Since restarting an application is essentially a roll-back of the application to its state at the time the checkpoint was taken, it would be ideal if the application's data files were also rolled-back to the state they were in at the time of checkpoint. Read-only files pose no problem since their

state does not change. Write-only files pose no problem assuming the task writing to it is deterministic (e.g., does not depend on real-time events). That is, checkpoint/restarting the task does not change what it writes to the file. In this case, the task will just overwrite, with the same values, whatever it wrote in the file after the checkpoint.

The real problem arises when files are accessed read-write. If a task reads some data from the file just before it was checkpointed, and then overwrites that data on the file with some other value right after, restarting the task would make it read the new value instead of the old one. In this case, the behavior of the task is changed, possibly affecting the behavior of the entire application. A similar problem arises when files are deleted.

A possible solution to this problem is to maintain a checkpoint of not only the task's process state, but also of its data files. However, considering data files can be of arbitrary size (possibly in the gigabyte range), it would not be practical to make a copy for checkpointing. A possible method for checkpointing read-write files is the use of lazy, copy-on-write checkpointing when non-idempotent file operations are done.

## 4.2 Terminal I/O

Terminal I/O is a common method for a user to pass parameters to an application. However, in the presence of migration and checkpoint/restart, an application's terminal connection is lost. Loss of terminal connection would likely change the behaviour of an application. It is therefore necessary to be able to maintain terminal I/O despite migration and checkpoint/restart.

There are two approaches to this problem. First, an application's stdin/out/err could be redirected to a proxy process that has terminal access. This approach has the nice property of maintaining interactive I/O between the application and the user.

The second and more simple approach is to run the application in batch mode, where input and output data are read from/written to specified files. Note that the traditional "`appl < in > out`" syntax will not work since redirection of input and output files is done by the command shell and will be lost on migration or checkpoint/restart. Instead of using the '`<`' and '`>`' redirection operators, MMPVM provides two optional flags for input and output file specification. For example, to run an application in batch mode, "`appl -iin -oout`" must be used. The effect is the same as if the '`<`' and '`>`' operators were used except that the MMPVM library now knows the names of the

input and output files. In this way, the terminal I/O problem could be treated as a file I/O problem, for which we have a partial solution. The '-i' and '-o' options are not passed on to the application.

### 4.3 GUIs

As currently implemented, the migration mechanism in MMPVM can only guarantee transparency as far as the PVM interface and some file I/O operations are concerned. Migrating tasks that use other location dependent facilities like sockets and shared memory outside the PVM library is bound to fail on migration and checkpoint/restart.

An implication of this restriction is that tasks that use the X-windows interface cannot be migrated nor checkpointed. The problem is that tasks that use X use sockets that the MMPVM system doesn't know about.

A possible way to avoid this restriction is through the use of proxy X servers. This is possible by supplying a stub X library which converts calls to the X libraries into PVM messages. From the task's point-of-view, it is calling the X primitives. However, the stub X library converts these calls to PVM messages to the proxy X-server which does the X calls in behalf of the task. A problem with this solution is with regards asynchronous X event handling where application functions (callbacks) have to be executed.

Nonetheless, in this manner, the task itself can be migrated. Note that the proxy X server still cannot be migrated. Another advantage is that the task doesn't have to be linked with the X libraries. Considering migratable tasks (as of the current implementation) are required to be statically linked, the non-inclusion of the X libraries will reduce the amount of data in the task, allowing for faster migration speeds.

### 4.4 Cross-Application Communication

A PVM application, as defined in MMPVM, is the set of tasks that have a common ancestor. That is, the set of tasks that belong to a single spawn tree. The concept of a PVM application is useful in checkpointing since this provides a convenient way of determining which tasks have to be synchronized for checkpointing, allowing different applications to be checkpointed independently.

However, within the current PVM framework, it is possible for multiple applications to communicate with each other. Cross-application communication is possible by using `pvm_tasks()` which returns information about all tasks in the PVM system, including

those of other applications. Another way is through the group server.

The implication of one application communicating with another is that the two applications should be treated as one big application, despite that fact that tasks in these applications belong to two distinct spawn trees. As such, the scheduler should be informed that these two applications should be scheduled together to maintain efficiency (e.g., gang scheduled) and also checkpointed together for correctness.

### 4.5 Multi-User Implementation

The current implementation of the multi-user PVM requires the pvmds to be running as root. While this solution works well, it does raise some security and administrative concerns. Security concerns in terms of allowing FTP'ed software to run as root and administrative concerns of requiring root privileges to install the software.

We are currently looking at other ways of allowing the scheduler to have global knowledge about all PVM tasks without requiring superuser privileges. One possibility is to maintain the per-user characteristic of PVM but modify the pvmds such they can all communicate with just one scheduler. This solution has a number of nice properties. First, it removes the requirement that the pvmds run as root at the same time leaving the scheduler with knowledge of all tasks in the system. Second, since each user has his/her own set of pvmds, they are isolated from each other.

### 4.6 Message Flushing Implementation

Because of the synchronous way we have implemented checkpointing, it is necessary to make sure that all messages between all tasks in the application are flushed. At checkpoint time, each task sends a special FLUSH message to all the other tasks, then waits till the FLUSH messages from all the other tasks are received. Since MMPVM guarantees message order by the use of sequence numbers, task A's receipt of the FLUSH message from task B implies receipt of all message previously sent by B to A, regardless of how the messages were sent (i.e., direct, indirect, multicast).

While this implementation is simple, the number of flush messages sent at checkpoint time is in the order of  $O(N^2)$ , where  $N$  is the number of tasks. By taking advantage of the fact that MMPVM messages are sequenced, the flushing protocol can be replaced by an  $O(N)$  algorithm. The new method requires a gather and broadcast operation of the last sent/received sequence number used by a task to all other tasks. With

this information, a task can determine if it has already received the last message sent to it by other tasks by comparing the last sent sequence number (from the gather/broadcast operation) with the sequence numbers of the messages it has already received.

## 4.7 Load Monitor Implementation

As mentioned above, we are currently using a very simple load monitor which allows the users to explicitly indicate whether a machine is available or not. Ideally, we would want to be able to detect machine availability automatically. There are systems like Condor, DQS, LSF, and PRM that already have this capability. By monitoring statistics managed by the OS like average run queue length, paging frequency, terminal activity, packet collisions, and numerous other data, it is possible to automatically determine if a machine is available (or usable).

But what does it really mean for a machine to be available? The simple case is when the owner is using it. In this case, the machine is considered unavailable regardless of whether the owner is running a large simulation or just reading mail. If the owner is not using it, however, then we have to rely on usage statistics. With all the statistics the OS kernel maintains about machine activity, there is a question of whether there is a minimal set of information that would accurately indicate if the machine is available for use or not. If so, what is this minimal set of information and how should they be interpreted? Answers to these questions is another active area of research within the Distributed Systems Research Group here at OGI.

## 4.8 Security

As with most, if not all, multi-user computing environments that span networks of workstations, security is an issue that must be addressed. We plan to investigate and integrate security features into MIST in several phases, with each phase building on the security provided by the previous phase. Security concerns of MIST users and programmers, workstation owners, and system administrators will be addressed.

The current MIST implementation, other than using standard Unix security facilities, does not offer any other security features of its own. Communication is unsecure, tasks are unauthenticated, and all tasks can access and communicate with all other tasks. Furthermore, the daemons run as root, and so represent a significant security hole due to the lack of authentication.

As mentioned in section 4.5, a possible alternative to our multi-user implementation is to allow each user to have his/her own daemons, all of whom talk to just one scheduler. This implementation will remove the requirement that the daemons be run as root. This will also isolate MIST users from each other, and remove the security problem presented by the highly privileged daemons.

The next step would be to add encryption, perhaps via SSL [7], to all communications. Encryption of all MIST network traffic would significantly complicate, if not entirely prevent, malicious eavesdropping on MIST data transmissions. This service could then be extended to add mutual authentication among all MIST tasks, possibly using a system like Kerberos [8]. Authentication would prevent unauthorized utilization of the MIST system and the resources to which it has access.

The final step would incorporate technology like Software Fault Isolation [9] to force strict adherence to the MIST API, thus preventing access to any unsecure, unauthenticated services.

## 5 Related Work

Condor [10, 11, 12], PRM [13], UPVM [14], DQS [15], Lsbatch [16], Fail-Save PVM [17], and DOME [18, 19, 20] are other software systems that support adaptive parallel application execution on a shared network of workstations. These systems either employ process migration, checkpointing, suspension, resumption, or a any combination of these to support adaptive execution. A common trait among these systems, MMPVM included, is that all these systems are implemented at user-level, requiring no special support from the hardware or the operating system.

Condor [10] is a software package that allows user applications unobtrusive access to machines on a shared network. Condor achieves load balance by executing user applications on idle or lightly loaded machines, at the same time remains unobtrusive to machine owners by migrating applications away from reclaimed machines.

Condor was initially designed for use with sequential applications. Migration of sequential applications is achieved by taking a core dump of the application and combining it with the application's executable file to create a checkpoint file. Extensions were then made to support parallel applications, PVM applications in particular [11, 12]. Resource management support for PVM applications include task scheduling, deletion, suspension and resumption. A notification service is

also provided to inform other tasks of the application that one of its tasks was deleted, suspended or resumed. Support for checkpointing and migration of parallel application is not (yet) supported.

Upon owner reclamation of a machine, tasks executing on that machine are suspended in the hope that the owner will only use the machine for a short time. If the owner remains active however, the tasks are deleted, and a notify message is sent to the other tasks to inform them of the deletion. While this resource management scheme works, it puts the burden of adapting to varying resource availability on to the application, and ultimately on the application developer.

By supporting transparent migration and checkpointing, MMPVM avoids having the application developer worry about how the application should adapt to varying resource availability. The developer can concentrate on the application itself, design it as if it were to run on a dedicated environment, and let the MMPVM system handle its execution on a shared environment. We hope to be able to integrate MMPVM with the Condor system in the future.

The Prospero Resource Manager (PRM) [13] is a software environment which provides a scalable and flexible resource management structure for execution of both sequential and parallel applications. PRM's and MMPVM's support for unobtrusive execution of parallel applications are functionally the same. They differ, however, in terms of implementation.

PRM has its own communication library which uses their Asynchronous Reliable Deliver Protocol (ARDP) for reliable delivery of sequenced packets over UDP. Support for PVM applications is through an interface library which translates PVM API calls to their library's API. Task migration uses Condor style checkpointing. Delivery of messages to migrated tasks is achieved through a timeout/retry mechanism that searches for the current location of the recipient task and resends the messages.

Since MMPVM is built on top of PVM itself, MMPVM supports the entire PVM API as defined by PVM. For example, setting the direct route option in MMPVM will actually create TCP connections. In contrast, since PRM uses a UDP based protocol, it will not create a TCP connection even if direct routing was specified. Another implementation difference is in how migration is realized. MMPVM transfers process state directly through the network. PRM, on the other hand, uses Condor style checkpointing which involves the creation of core dumps and checkpoint files. From our experience with both types of migration schemes,

our method is approximately 10x faster than the core dump method.

UPVM [14] is another software system, very similar to MMPVM, that enables transparent migration of PVM tasks. Unlike MMPVM where each task is a Unix process, however, tasks in UPVM are implemented as User-Level Processes (ULPs). ULPs are like Unix processes except that there is no protection between multiple ULPs allowing for minimal context switching cost and there can be multiple ULPs in one Unix process. Unlike threads, however, each ULP has its own data and heap space. By having its own data and heap space, in addition to its stack, each ULP is independently migratable.

The primary purpose of UPVM is to address the coarse-grained distribution granularity present in MMPVM. MMPVM migrates tasks at the level of Unix processes. Since ULPs are "smaller" processing entities, they have the potential for achieving faster migration speeds and better load balance.

As currently implemented, UPVM has two main restrictions. First, it only runs SPMD programs. Since each ULP within a Unix process shares its text with the other ULPs (in the same process), the PVM application has to be designed in SPMD style. Second, since multiple ULPs share the address space of a single Unix process, there is a limit on the number of ULPs the application can have. This limit depends on the size of the virtual address space of the Unix process and the memory requirements of each ULP.

DQS [15] and LSF (successor of Lsbatch [16]) are two other software systems that mainly support load balancing of batch applications on a network of workstations. These systems support execution of parallel applications to a limited extent. Recent releases of DQS 3.1.2 and LSF 2.1 are said to have better support for parallel applications, but the extent of this support unknown to us at this time.

Fail-Safe PVM [17] is an extension to PVM that implements transparent application checkpointing and restart. Just like the current implementation of MMPVM's checkpoint/restart facility, Fail-Safe PVM uses synchronous checkpointing and messages are explicitly flushed at checkpoint time.

The difference between the checkpoint/restart facilities of Fail-Safe PVM and MMPVM is that MMPVM can be selective about which tasks should be checkpointed/restarted. For example, if multiple applications are executing, one application could be checkpointed independently of the other. Along with the ability to independently checkpoint applications is the special processing that has to be done when check-

pointing the group server. It has to be stated, however, that this difference between Fail-Safe PVM and MMPVM was essentially brought about by our goal of making PVM multi-application and multi-user, a goal Fail-Safe PVM wasn't designed for.

DOME [18, 19] is a computing environment that supports heterogeneous checkpointing through the use of C++ class abstractions. DOME also supports dynamic load balancing through transparent data redistribution. Checkpoints are generated at a "high-level" based on data structures and variables that the application developer defines as part of the DOME environment. This allows for 1) efficient checkpointing since DOME knows exactly what data needs to be saved, and 2) heterogeneous checkpoints since DOME knows of the data types of the data structures it has to save and thus could save them in XDR format, for example. Another big difference between DOME and MMPVM is that DOME doesn't save the state of the stack. Instead, it relies on a very structured programming model where the point at which the application was checkpointed should be accessible through a "goto" statement from `main()`. A way of getting around this restriction is proposed in [20] by the use of a preprocessor that annotates the application with the necessary statement labels and goto calls.

Aside from these software systems, support for adaptive execution on a shared environment is also available from systems such as Sprite [21, 22], Mosix [23, 24], V [25], Mach [26] and Chorus [27]. The difference between these systems from MMPVM and those mentioned previously is that these systems are implemented at the operating system level. While these systems can handle most of the problems associated with user-level implementations (e.g., total migration transparency and efficient task checkpointing and migration), they are not as portable/available compared to user-level implementations.

## 6 Future Work

It is obvious that a lot of work still has to be done before the MIST project can be realized. In one way or another, the issues raised in section 4 have to be addressed. In this regard, we are taking an incremental approach by addressing parts of the problem one at a time rather than trying to find a "solve-it-all" solution.

Second, we would like to be able to integrate our software with software systems already available from other research groups such as Condor, PRM, DQS, AVS [28], XPVM [29], PIOUS [30], Ptools [31], etc.

and, of course, with the official PVM release.

## 7 Summary

This paper presented our prototype for an intelligent scheduling and resource management system for running PVM applications. By making use of MMPVM's transparent task migration and checkpointing capabilities, combined with machine information available from load monitors, our scheduler is capable of executing a PVM application in a shared environment, adapting to varying system load, and making unobtrusive use of idle-cycles. The scheduling algorithm used in the scheduler makes use of gang scheduling and processor doubling to minimize resource utilization without affecting application performance.

The work presented in this paper is only a part of our vision of a unified distributed/parallel application development and execution environment (figure 1). The ultimate goal of the project is to create an environment that integrates the various components, possibly from other systems, that can be used by parallel application developers and users. By taking an open systems approach, the project promotes "plug-and-play" integration of these various components through well defined interfaces. These interfaces enable each component to be developed and used independently.

## References

- [1] A. L. Beguelin, J. J. Dongarra, A. Geist, and R. J. Manchek 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] Jeremy Casas, Dan Clark, Ravi Konuru, Steve Otto, Robert Prouty, and Jonathan Walpole. MPVM: A migration transparent version of PVM. Technical Report CSE-95-002, Dept. of Computer Science and Engineering, Oregon

- Graduate Institute of Science & Technology, February 1995.
- [5] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. PVM 3 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1994.
- [6] Khaled Al-Saqabi, Steve W. Otto, and Jonathan Walpole. Gang scheduling in heterogenous distributed systems. Technical Report CSE-94-023, Dept. of Computer Science and Engineering, Oregon Graduate Institute of Science & Technology, August 1994.
- [7] Kipp E. B. Hickman. The SSL protocol. RFC draft specification, available at <http://home.netscape.com/newsref/std/SSL.html>.
- [8] J. Steiner, C. Neuman, and J. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Usenix Winter Conference*, pages 191–202, Berkeley, CA, February 1988.
- [9] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Software fault isolation. In *Fourteenth ACM Symposium on Operating System Principles*, volume 27, pages 203–216, December 1993.
- [10] Michael J. Litzkow, Miron Livny, and Matt 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.
- [11] Jim Pruyne and Miron Livny. Providing resource management services to parallel applications. In J. Dongarra and B. Tourancheau, editors, *2nd workshop on Environments and Tools for Parallel Scientific Computing*, pages 152–161, 1995.
- [12] Jim Pruyne and Miron Livny. Parallel processing on dynamic resources with CARM. In *Proceedings of the Workshop on Job Scheduling for Parallel Processing, International Parallel Processing Symposium '95*, April 1995.
- [13] B. Clifford Neuman and Santosh Rao. The Prospero Resource Manager: A scalable framework for processor allocation in distributed systems. *Concurrency: Practice and Experience*, 6(4):339–355, June 1994.
- [14] Ravi Konuru, Jeremy Casas, Steve Otto, Robert Prouty, and Jonathan Walpole. A user-level process package for PVM. In *1994 Scalable High-Performance Computing Conference*, pages 48–55. IEEE Computer Society Press, May 1994.
- [15] T. Green and J. Snyder. DQS, a distributed queuing system. Technical report, Supercomputer Computations Research Institute, Florida State University, April 1993.
- [16] J. Wang, S. Zhou, K. Ahmed, and W. Long. Lsbatch: A distributed load sharing batch system. Technical Report CSRI-286, Computer Systems Research Institute, University of Toronto, Toronto, Canada, April 1993.
- [17] Juan Leon, Allan Fisher, and Peter Steenkiste. Fail-safe PVM: A portable package for distributed programming with transparent recovery. Technical Report CMU-CS-93-124, Carnegie Mellon University, February 1993.
- [18] Adam Beguelin, Erik Seligman, and Michael Starkey. Dome: Distributed object migration environment. Technical Report CMU-CS-94-153, Carnegie Mellon University, May 1994.
- [19] Erik Seligman and Adam Beguelin. High-level fault tolerance in distributed programs. Technical Report CMU-CS-94-223, Carnegie Mellon University, December 1994.
- [20] Jose Nagib Cotrim Arabe, Adam Begueline, Bruce Lowekamp, Erik Seligman, Mike Starkey, and Peter Stephan. Dome: Parallel programming in a heterogeneous multi-user environment. Submitted as a Technical Paper for Supercomputing '95.
- [21] Fred Douglass and John Ousterhout. Process migration in the Sprite operating system. In *Proceedings of the 7th IEEE International Conference on Distributed Computing Systems*, pages 18–25, Berlin, West Germany, September 21–25 1987.
- [22] Fred Douglass and John Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software — Practice & Experience*, 21(8):757–785, August 1991.
- [23] Amnon Barak and Ami Litman. MOS — A multi-computer distributed operating system. *Software — Practice & Experience*, 15(8):725–737, August 1985.

- [24] Amnon Barak, Shai Geday, and Richard G. Wheeler. *The MOSIX Distributed Operating System – Load Balancing for Unix*. Lecture Notes in Computer Science. Springer-Verlag, 1993.
- [25] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable remote execution facilities for the V-System. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 2–12, Orcas Islands, Washington, December 1–4 1985.
- [26] Dejan S. Milojicic, Wolfgang Zint, Andreas Dangel, and Peter Giese. Task migration on the top of the Mach microkernel. In *MACH III Symposium Proceedings*, pages 273–289, Santa Fe, New Mexico, April 19–21 1993.
- [27] M. O’Connor, B. Tangney, V. Cahill, and N. Harris. Microkernel support for migration. Submitted to *Distributed Systems Engineering Journal*, December 1993.
- [28] G. Cheng, G. Fox, K. Mills, and Marek Podgorny. Developing interactive PVM-based parallel programs on distributed computing systems within AVS framework. In *3rd Annual International AVS Conference, JOIN THE REVOLUTION: AVS ’94*, Boston, MA, May 1994.
- [29] James A. Kohl and G. A. Geist. XPVM: A graphical console and monitor for PVM. In *2nd PVM User’s Group Meeting*, Oak Ridge, TN, May 1994.
- [30] S. Moyer and V. S. Sunderam. PIOUS: A scalable parallel i/o system for distributed computing environments. In *1994 Scalable High-Performance Computing Conference*, pages 71–78. IEEE Computer Society Press, May 1994.
- [31] William Gropp and Ewing Lusk. Scalable Unix tools on parallel processors. In *1994 Scalable High-Performance Computing Conference*, pages 56–62. IEEE Computer Society Press, May 1994.