MPVM: A Migration Transparent Version of PVM

Jeremy Casas, Dan Clark, Ravi Konuru,
Steve Otto, Robert Prouty, Jonathan Walpole
{casas,dclark,konuru,otto,prouty,walpole}@cse.ogi.edu

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

February 1995

Abstract

Parallel Virtual Machine (PVM) is a widely-used software system that allows a heterogeneous set of parallel and serial UNIX-based computers to be programmed as a single distributed-memory parallel machine. In this paper, an extension to PVM to support dynamic process migration is presented. Support for migration is important in general-purpose workstation environments since it allows parallel computations to co-exist with other applications, using idle-cycles as they become available and off-loading from workstations when they are no longer free. A description and evaluation of the design and implementation of the prototype Migratable PVM system is presented together with some performance results.

1 Introduction

PVM [1, 2, 3] is a 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 distributed-memory virtual computer. Such a system allows the computing power of widely available, general-purpose computer networks to be harnessed for parallel processing. With the rapid advances in workstation performance, such networks already provide a viable and affordable alternative to expensive special-purpose super-computers.

General-purpose workstation networks have certain key characteristics that must be considered when they are to be used for parallel processing. First, the collective resources of the network are often shared by a potentially large number of users running a wide range of applications. Second, despite the high level of sharing, the concept of ownership is frequently present. In particular, individual workstations, while available across the network, are invariably owned by some specific user. Workstation owners are often willing to allow others to access their workstation when it is idle, but expect dedicated access the rest of the time. Since most workstations are idle most of the time [4], the key to harnessing the full power of such systems lies in gaining access to these idle cycles.

For PVM to gain unobtrusive access to idle cycles, it must be able to (a) recognize when a workstation becomes available for it to use, (b) recognize when a workstation ceases to be available to it, and (c) migrate processes between workstations so that work can be assigned to newly available workstations and off-loaded from workstations that are being reclaimed by their owners. Automatic and timely off-loading of processes requires PVM to be extended to support dynamic process migration.

In this paper, Migratable PVM (MPVM), an extension of PVM which allows parts of the parallel computation to be suspended and subsequently resumed on other workstations is presented. There were three key goals under consideration in the design of MPVM. First, migration had to be transparent to both application programmer and user. Neither the programmer nor the user needs to know that portions of the application are migrating. Second, source-code compatibility with PVM had to be maintained. Source-code compatibility would allow existing PVM applications to run under MPVM without, or at least with minimal, modification. Lastly, MPVM had to be as portable as possible.

The remainder of the paper is organized as follows. Section 2 gives an overview of PVM and the problem addressed by this work. Section 3 outlines the design and implementation of MPVM, and is followed by performance results in section 4. Related work is presented in section 5, a qualitative discussion of the design and implementation in section 6, and conclusions and future work in section 7.

2 Background

MPVM is based on PVM 3.3.4 as released from Oak Ridge National Laboratory and is part of the larger Concurrent Processing Environment, an on going research effort [3]. This section presents an overview of the PVM system and the problems that had to be addressed to support task migration.

2.1 PVM Overview

The PVM system consists of a daemon process called the *pvmd* running on each host on a network of workstations and a run-time library called the *pvmlib* linked into each application process (figure 1). Each pvmd is assigned a unique host ID or hid. The pvmlib defines a suite of PVM primitives that presents a "message-passing parallel machine" user-interface to the application.

A PVM application is composed of Unix processes linked with the pvmlib. These processes, called tasks in PVM, communicate with each other via message-passing primitives found in the pvmlib. Just like the pvmds, each task is assigned a task ID or tid which uniquely identifies each task in the virtual machine. These tids are used to designate the source and destination tasks for messages (i.e., messages are addressed to tasks, not to ports or mailboxes).

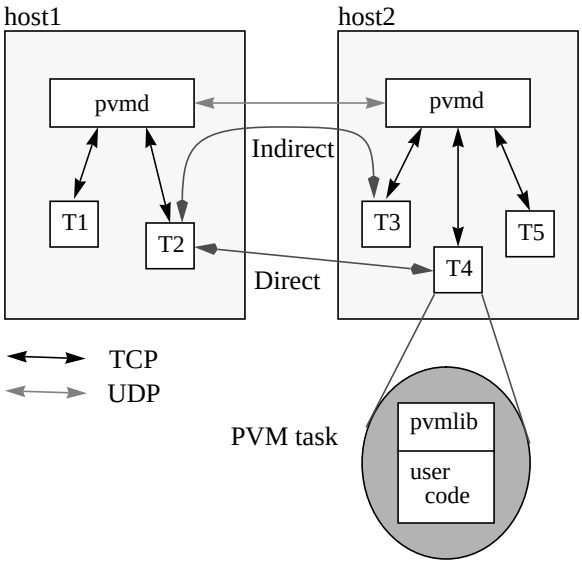

Figure 1. PVM System. PVM is composed of daemons (*pvmds*) running on each host of the virutal machine and a run-time library (*pvmlib*) linked into every task.

Messages passed within the PVM system can be categorized into system messages and application messages. System messages are used exclusively by PVM to manage the virtual machine and perform application code requests (e.g., spawn a new task, get information about the virtual machine, etc.). The application code is not aware of these messages. Application messages on the other hand are used exclusively by the PVM application.

PVM provides two routing mechanisms for application messages: indirect and direct routing. The choice of routing mechanism to use is controlled by the application code. By default, messages are routed indirectly. Using indirect routing, as illustrated in figure 1, a message from task T2 to T3 passes through T2's local pvmd (pvmd on host1), through T3's local pvmd (pvmd on host2), and finally to T3. Pvmd-to-pvmd communication uses UDP socket connections while task-to-pvmd communications use a TCP socket connection which is established during task start-up. In direct routing, a message from task T2 to T4, also illustrated in figure 1, uses a TCP socket connection between T2 and T4, by-passing the pvmds altogether. TCP connections between tasks are created "on-demand". Only when tasks that have set their routing option to use direct routing start communicating with each other are TCP connections established.

An important aspect to remember when using PVM is the message ordering semantics it provides. PVM guarantees that messages sent from one task to another are

received in the same order they were sent. The importance of recognizing this "guarantee" is that there are PVM applications that take advantage of this message-ordering semantics. Hence, new versions of PVM such as MPVM should maintain the same semantics.

Lastly, in PVM 3.3.4, it is possible to designate a special task as the resource manager. The resource manager, also called the global scheduler (GS) in this paper, embodies decision making policies [5] such as task-to-processor allocation for sensibly scheduling multiple parallel applications. Using a GS makes it convenient to experiment with different scheduling policies. In MPVM, the interface between the pvmds and the GS has been extended to accommodate task migration, allowing the GS to use dynamic scheduling policies.

2.2 PVM task migration: the problem

Task migration is the ability to suspend the execution of a task on one machine and subsequently resume its execution on another. A major requirement for task migration is that the migration should not affect the correctness of the task. Execution of the task should proceed as if the migration never took place. To ensure the "transparency" of the migration, it is necessary to capture the state of the task on the source machine and reconstruct it on the target machine.

The state of a task can be viewed in two ways: its state as a Unix process and its state as a task of a PVM application. From the point of view of the operating system (OS), a task is just a single process. As such, its state includes the processor state, the state held by the process, the state held by the OS about the process, and the state held by the process about the local OS. The processor state includes the contents of the machine registers, program counter, program status word, etc. This information defines exactly where the task was executing prior to migration, and consequently, where execution should resume upon restart on the target machine. The state held by the process itself include the contents of its text, data (static and dynamic), and stack segments. The state held by the OS for the process include signal information (e.g., blocked signals, pending signals), open files, and socket connections to name a few. Other less obvious state information held by the OS include page table entries, controlling terminals, and process relationship information (e.g., parent/child process relationship and process groups). OS state held by the process include file descriptors, process IDs, host name, and time. These are state information, known to the process, that are only valid in the context of the local execution environment (local OS and host).

From the point-of-view of PVM, a task is one of a set of tasks that makes up an application. In this regard, a task's state includes its tid and the messages sent to/from that task. Regardless of migration, each task should be referred to using the same tid, no message should be lost, and all messages should be received in the correct order (as defined by PVM).

Thus, the problem addressed by MPVM is how to capture and reconstruct the state information so that tasks can to be migrated from one machine to another without affecting the correctness of the entire application.

3 Design and Implementation

In this section, the design and implementation of MPVM is described. In order to support task migration, both the pvmd and pvmlib had to be modified. The modifications made were also driven by the goals of source code compatibility, portability, and migration transparency. To ensure source code compatibility, the modifications had to maintain the same function calls, parameters and semantics, as provided by PVM. To maximize portability, the migration mechanism had to be implemented at user-level, using facilities available through standard Unix library routines and system calls. Migration transparency is addressed by modifying the pvmd and pvmlib such that the migration could occur without notifying the application code and by providing "wrapper" functions to certain system calls. A more complete evaluation of these goals are presented in a later section.

3.1 Application start-up

The primary interface to the migration mechanism in MPVM is through the signal mechanism provided by Unix. That is, task migration is initiated using a migration signal sent from the pvmd to the migrating task. The migrating task should have a migration signal handler installed to catch the migration signal. At this point, it is only important to know that a signal handler has to be installed for migration to work. The function of the migration signal handler will be discussed in the next section.

To avoid explicitly modifying the source code of the PVM application to install the signal handler, the pvmlib defines its own **main()** function which executes the necessary initialization and then calls a function called **Main()**. When the application program is compiled, its **main()** function is "renamed" to **Main()** using "C" macro substitution facilities available through the compiler (e.g., *-Dmain=Main* flag). Thus, when the application code is

linked with the pvmlib, the resulting executable will have the pvmlib's **main()** as the entry point, allowing execution of the migration initialization code prior to the execution of the application's code.

While this solution is simple, not to mention inherently portable, it will fail when combined with other systems that use the same "trick" for executing code prior to the application's **main()**. The alternative, however, is to define a customized version of the start-up code, usually crt0.o ("C" Run-Time object module), which is more troublesome to create, maintain, and port.

3.2 Migration Protocol

Once the application is up and running, it executes just like a traditional PVM application would, until a task has to migrate. There are a number of reasons for a task to migrate: excessively high machine load, machine reclamation by its owner, a more suitable machine becomes available, etc. Regardless of the rationale for migration, the same migration mechanism can be used.

A migration protocol is used to facilitate the migration. The migration protocol is divided into four stages as shown in figure 2. While the first stage addresses "when" migration occurs, the last three stages correspond exactly to the main components of migration: state capture, transfer, and re-construction.

An important component of the migration protocol is what is collectively called Control Messages. These control messages or CMs are special system messages added to the pvmds and the pvmlib for the primary purpose of managing task migration. Just like other system messages, these control messages are invisible to the application code. There are different kinds of CMs, each of which will be discussed in the following sections.

3.2.1 Migration event

The migration of a task is triggered by a migration event. This event triggers the GS which determines whether or not tasks have to be migrated. If so, it also decides which tasks to migrate and to where.

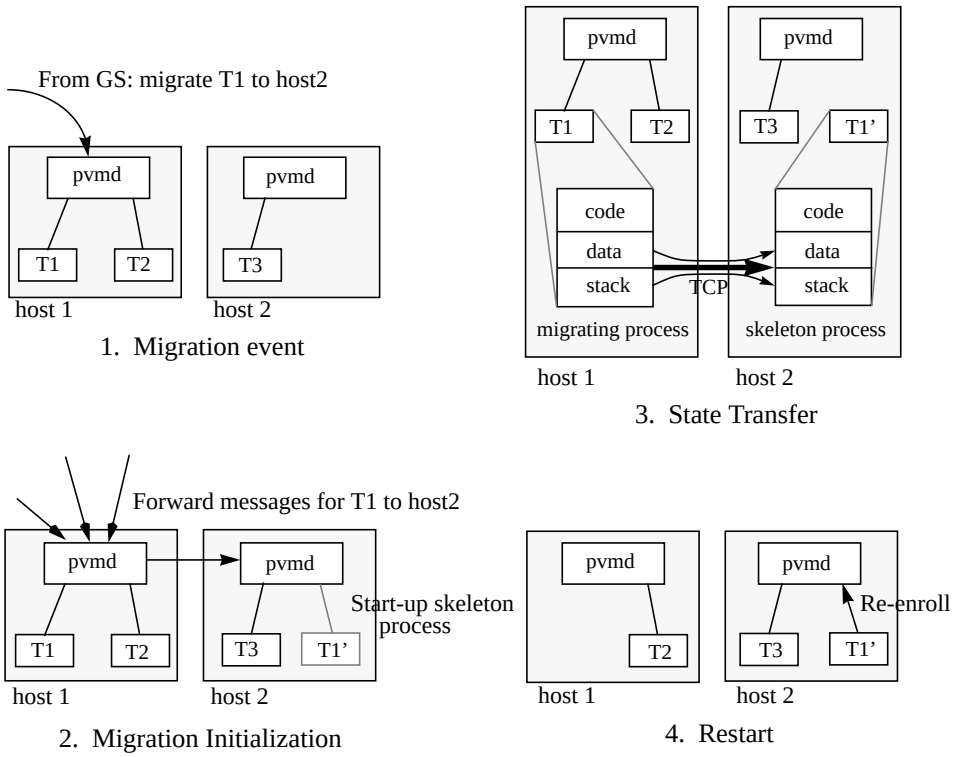

Figure 2. The migration protocol, illustrating the stages involved in migrating task T1 from host 1 to host 2.

If the GS decides to migrate a task, an *SM_MIG* CM (SM stands for Scheduler Message) is sent by the GS to the pvmd on the host where the task to be migrated is currently executing. This *SM_MIG* CM contains a tid and an hid, indicating the task to be migrated and the destination host respectively. For brevity, the task to be migrated shall be referred to as Mtask, the pvmd on the host where Mtask will be migrating from as Spvmd, and the pvmd on the destination host as Dpvmd.

3.2.2 Migration Initialization

Upon receipt of an *SM_MIG* CM, the Spvmd verifies that the tid belongs to a locally executing task and that the hid refers to a valid host (not itself). If either of the tid/hid is invalid, a PvmNoTask/PvmNoHost error code is sent back to the GS via an *SM_MIGACK* CM.

Migration initialization is divided into two components which occur in parallel. The first component, local initialization, involves "priming-up" Mtask for the state transfer. The second component, remote initialization, involves the creation of a "skeleton process" that will be the recipient of the state information to be transferred (figure 3).

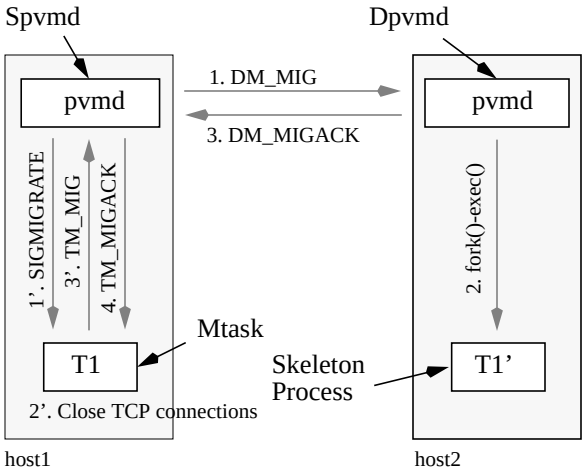

Figure 3. Migration Initialization. Local initializations (1', 2', and 3') and remote initializations (1, 2, and 3) occur in parallel and 'sync' at step 4 which is the final message inidicating that process state transfer can proceed.

Local initialization begins when a SIGMIGRATE signal is sent from the Spvmd to Mtask (step 1' in figure 3). This signal is caught by the migration signal handler installed by Mtask during its start-up (recall the application start-up discussion). The advantage of using a migration signal handler is two-fold: first, it allows for asynchronous task migration, and second, it is the main component used in capturing the processor state. When the signal handler is invoked, the OS automatically stores a copy of the processor state at the time when the process was interrupted in the user stack. This saved processor state is used to restore the state of execution of the process/task upon return from the signal handler. Currently, SIGUSR1 is used as the SIGMIGRATE signal.

To prepare for the process state transfer in the next stage, the migration signal handler in Mtask flushes all messages in the TCP socket connections it has with other tasks (used in direct message routing), and then closes these connections (step 2'). It is necessary to flush these TCP socket connections to avoid losing any message that may be buffered in these sockets. The details of how the connections are flushed and closed will be discussed in a later section. Note that the tasks with which Mtask had a direct TCP connection continue executing as they normally would. In the event they send a message to Mtask, the message will automatically be forwarded through the pvmds.

Once the TCP connections have been flushed and closed, Mtask sends a *TM_MIG* CM (TM stands for Task Message) to Spvmd (step 3') to tell the Spvmd that local initialization is complete. Mtask then blocks and waits for a *TM_MIGACK* CM from the Spvmd.

While the local initialization component is executing in the source machine, remote initialization is proceeding simultaneously on the destination machine. Remote initialization is triggered by a *DM_MIG* CM (DM stands for Daemon Message) from Spvmd to Dpvmd (step 1, no prime). This CM informs the Dpvmd that a task will be migrating to it. Information about the migrating task such as its tid, executable file name, parent task's tid, etc. is passed along in this CM. The name of the executable file from which the migrating task was started is particularly important since the same executable file should be used to start a "skeleton process" (step 2). The executable file is assumed to be accessible from the destination machine. The skeleton process provides the infrastructure to which process state can be transferred and will eventually be executing in the context of Mtask.

State transfer has three requirements: the source of the state, the recipient of the state, and the medium through which the state will be transferred. The first two components are satisfied by Mtask and the skeleton process respectively. For the transfer medium, a TCP connection, to be established at process state transfer time, is used. For the TCP connection to be established, it is necessary

that the skeleton process have a TCP socket to which Mtask can "connect" to. The decision of associating a TCP socket with the skeleton process and making Mtask initiate the connection was made for convenience rather than some hard requirement. To assign a socket to the skeleton process, before Dpvmd starts the skeleton process, it first creates a socket and binds it to a port address. Following the semantics of **fork()**/**exec()**, the skeleton process automatically inherits the socket from the Dpvmd.

In addition to making the skeleton process inherit the socket, special arguments are also passed to the skeleton process. These special arguments causes the skeleton process to execute "restart code". Recall that at application start-up, some migration initialization code is first executed prior to executing the application's code. Part of the migration initialization code is to test whether the process has to execute as a skeleton process or not, based on the arguments passed to it. If the process was started as a skeleton process, it will wait for a connection on the socket that it inherited from the Dpvmd. If not, it executes the code of the application.

For Mtask to be able to connect to the socket waited on by the skeleton process, Mtask must know the port address the socket is bound to on the destination machine. This port address is known to the Dpvmd. To send the port address to Mtask, the Dpvmd sends a *DM_MIGACK* CM to the Spvmd (step 3) containing the port address and an error code. If the error code is zero, then the port address is valid and can be used by Mtask to connect to the skeleton process. A non-zero error code indicates that something went wrong during remote initialization and that migration cannot proceed. Possible error codes are PvmNoFile and PvmOutOfRes. A PvmNoFile error code means that the executable file name of the migrating task was not found on the destination machine. A PvmOutOfRes error code means that there wasn't enough resources on the destination machine to start the skeleton process. This error could be caused by several factors such as inability to create more sockets, inability to **fork()** another process, etc. A non-zero error code causes the Spvmd to send the GS an *SM_MIGACK* CM containing the error code, similar to what it would have done given an invalid tid or hid from an *SM_MIG* CM. Sending these error codes back to the GS allows the GS to keep track of unsuccessful migrations, giving it an up-to-date view of the state the PVM system.

The last part of this stage is for the Spvmd to send a *TM_MIGACK* CM to Mtask (step 4). Recall that at the end of the local initialization, Mtask blocks waiting for this message. But before the Spvmd can send this CM to Mtask, it must be sure that both the local and remote initializations have completed. Completion of local and remote initializations is indicated by receipt of both the *TM_MIG* CM from Mtask and the *DM_MIGACK* CM from the Dpvmd.

The *TM_MIGACK* CM sent to Mtask contains three items: an error code, the IP address of the destination machine, and the port address of the socket to connect to on the destination machine (the one the skeleton process is waiting on). If the error code is zero, then the migration protocol proceeds to the next stage. If the error code is non-zero (for reasons mentioned above), the migration is aborted and Mtask simply returns from the migration handler and continues its execution prior to getting interrupted by the SIGMIGRATE signal.

Conceptually, this stage of the protocol is simple. Unfortunately, the same cannot be said for the actual implementation. In particular, note that while within the migration signal handler, the CMs *TM_MIG* and *TM_MIGACK* are sent and received respectively. Sending and receiving these CMs requires the migration signal handler to use routines in the pvmlib. However, the pvmlib is not re-entrant. If the migration signal handler happened to be invoked while Mtask was executing within the pvmlib, the migration signal handler's use of pvmlib routines could corrupt data structures in the pvmlib, leading to unpredictable results.

The obvious solution is to make the pvmlib re-entrant. Making the pvmlib re-entrant, however, would require a complete re-work of the library. The simpler but not so elegant approach of synchronizing migration signal handler invocation with the task's execution in the pvmlib is used. One way of achieving this synchronization is to block the SIGMIGRATE signal whenever task execution enters the pvmlib. The approach was tried and worked as expected. Unfortunately, blocking and unblocking signals require system calls that incur a significant amount of overhead.

The solution used, which gives the same result as that of blocking signals but with much less overhead, is to set an *IN_LIB* flag whenever task execution enters the pvmlib. When the migration signal handler is invoked, this *IN_LIB* flag is checked first. If the flag is not set, the migration signal handler can safely use pvmlib routines. If it is set however, the migration signal handler sets another flag called the *IS_MIG* flag, indicating that the signal occurred, and returns. When execution of the task leaves the pvmlib, in addition to clearing the *IN_LIB* flag, the *IS_MIG* flag is checked. If the *IS_MIG* flag is set, the task gener-

ates a SIGMIGRATE signal to itself. Sending the signal to itself "simulates" the situation when the signal was first received, except that this time, the task is already outside the pvmlib.

A problem arises when execution of the task blocks inside the pvmlib such as when the user code calls a **pvm_recv()** and the desired message has not yet arrived. This situation is undesirable since the migration signal handler will not get invoked for an indefinite amount of time, preventing the task from migrating. To resolve this problem, all PVM routines that could block for an indefinite amount of time were rewritten so that they blocked outside the pvmlib. That is, modifications were made such that instead of having the routines block deep inside the pvmlib as they would in the original pvmlib, they now block on the "surface" of the pvmlib. This change is sufficient to allow the pvmlib routines used in the migration signal handler to be executed without running into reentrancy problems.

3.2.3 Process State Transfer

Reaching this stage of the migration protocol implies that the skeleton process was successfully started and that Mtask has received the *TM_MIGACK* CM containing the destination host's IP address and the port address of the socket the skeleton process is waiting on.

Before the state of Mtask is transferred, Mtask first detaches from the local pvmd (Spvmd in this case) using **pvm_exit()**. This call closes the TCP socket connection Mtask has with its local pvmd. Messages in the pvmlib that have not yet been received by the application remain intact in the task's data space.

As mentioned above, migration involves capturing the process' state (text, data, stack, and processor context), transferring it to another host, and reconstructing it. The text of the process can be taken from the executable file from which the process was started. It is for this reason the skeleton process is started from the same executable file from which Mtask was started. Using the same executable file automatically "migrates" the text. The data and stack, however, have to be read directly from Mtask's virtual memory. As for the processor context, recall that this has already been saved in the stack when the migration signal handler was invoked. By performing the state transfer while within the migration signal handler, coupled with the ability to transfer/restore the stack correctly, the processor context is preserved.

The processor context saved due to the invocation of the migration signal handler contains information regarding where execution should resume in the user's code. However, if migration is to occur within the signal handler, a second set of processor context information is needed to determine where execution should resume inside the signal handler. Correctly resuming execution inside the migration signal handler is necessary for the signal handler be able to "return" correctly and restore the process context saved when the signal handler was invoked. For this purpose, a **setjmp()** is called within the migration signal handler just before the actual state transfer. A similar approach is taken in Condor [6].

After calling **setjmp()**, Mtask creates a TCP socket and using the IP address and the socket port address from the *TM_MIGACK* CM, establishes a connection with the skeleton process on the destination host. It is through this TCP connection that the data and stack of Mtask is transferred.

3.2.4 Restart

After sending all the necessary state information to the skeleton process, Mtask terminates. It is at this point where Mtask is officially removed from the source host. The skeleton process, after receiving Mtask's state, assimilates it as its own. This assimilation of state is done by placing the received data and stack state in their appropriate place in the skeleton process' virtual address space. A temporary stack is used, again using the signalling facility, while restoring the state of the real stack to avoid corrupting its contents. After restoring all the state information from Mtask, a **longjmp()** is done using the buffer saved from the **setjmp()** call in the state transfer stage. This **longjmp()** causes execution to "go back" into the migration signal handler just as it was in Mtask at the time the **setjmp()** was called. It is at this point that the skeleton process starts executing in the context of the Mtask.

Before the skeleton process could re-participate as part of the application, it first has to re-enroll itself with the local pvmd (Dpvmd in this case) using the **pvm_mytid()** routine. By re-enrolling to the PVM system, the skeleton process officially becomes an MPVM task, at the same time re-establishing its indirect communications route with the other tasks. As for the TCP connections that were closed prior to the state transfer, note that direct connections are established "on demand" in PVM. That is, only when a message is first sent between two tasks (which have set their routing mode to use direct routing) is the TCP connection established. By closing down the TCP connections in such a way that the tasks involved "think" that there was never a connection, direct connections with

the just-migrated task will automatically be re-established, using the protocol provided by PVM, once messages start flowing between them again.

Lastly, though no longer technically part of the restart stage, the Dpvmd sends a *SM_MIGACK* CM to the GS containing an error code of zero. This CM informs the GS that the migration was successful and that the migrated task is again up and running.

Figure 4 shows the timeline of the migration protocol. Note that the migration protocol only involves the migrating task, the source pvmd, and the destination pvmd. Multiple migrations can occur simultaneously without interfering with each other, even if they have overlapping pvmds.

3.3 Closing Direct TCP connections

As mentioned in the migration initialization stage above, the TCP socket connections Mtask has with other tasks have to be flushed and closed prior to migration. These TCP socket connections are used for direct routing between Mtask and the other tasks. The TCP connections are flushed to avoid loss of any un-received message.

Flushing and closing these TCP socket connections is not as simple are just reading everything that could be read from the socket and then closing them. It is possible that messages are still in-transit and thus not yet available for reading. It is also possible that the peer task (the task at the other end of the connection) is just about to send a message. In either instance, the fact that nothing can be read from the TCP socket connection does not imply that there wouldn't be any in the future.

To ensure that there are no messages in the connection, in-transit or in the future, it is necessary for Mtask to explicitly inform the peer task of its intention of closing the connection and get an acknowledgment from the peer task that it will no longer send messages through that connection. To inform the peer task of the intention of closing the connection, Mtask sends an out-of-band (OOB) data, using the MSG_OOB flag for the **send()** system call, through the TCP connection. The OOB data causes a SIGURG signal at the peer task. Using this method of inform-

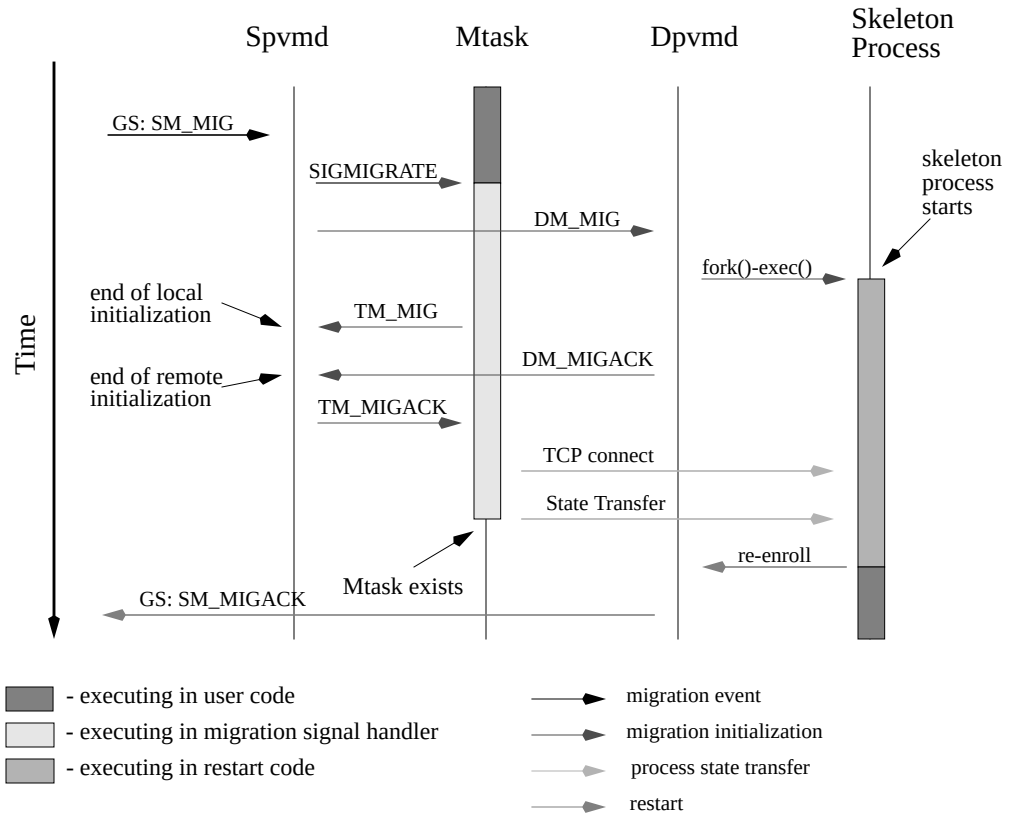

Figure 4. Migration protocol timeline.

ing the peer task of the connection closure has a number of advantages. First, it enables the peer task to respond to the socket connection closure immediately. Minimizing the time it takes to take down Mtask's TCP socket connection is necessary to minimize the time it takes to migrate Mtask. Second, by testing for "exception conditions" using **select()**, this method provides enough information for the peer task to know which socket, assuming it also has TCP socket connections with other tasks, is being closed. And lastly, this method involves only Mtask and the peer task which helps minimize the overhead involved in closing the connection.

The exact protocol used is illustrated in figure 5. Another feature of TCP socket connections that the protocol uses is the ability to close only one channel of the connection with the **shutdown()** system call. TCP socket connections can be pictured as two uni-directional pipes or channels. Using the **shutdown()** system call, it is possible to close the TCP socket connection one pipe or channel at a time. The **close()** system call closes both channels at once.

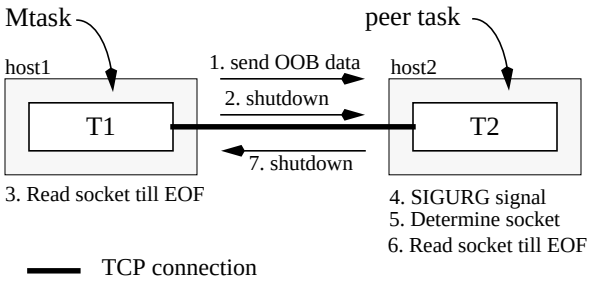

Figure 5. TCP socket connection closure protocol.

Initially, Mtask (T1) sends the OOB data to the peer task (T2). It then calls **shutdown()** to close the channel from T1 to T2, and proceeds to read the socket (i.e., reading the T2 to T1 channel) until it reads an end-of-file (EOF). The OOB data in the mean time causes a SIGURG signal to be generated at the peer task, which in turn invokes a SIGURG signal handler. The SIGURG signal handler at the peer task first determines which socket caused the SIGURG signal using the **select()** system call, and then reads in all it can from the socket until it reads an EOF. The reading of the socket until an EOF is detected, in effect, flushes any un-received messages sent by Mtask. The EOF is guaranteed to be detected due to the **shutdown()** call in step 2. After detecting the EOF, the signal handler issues a **shutdown()** on the T2 to T1 channel. At this point, Mtask is still reading the socket waiting for an EOF on the T2 to T1 channel. Once Mtask reads the EOF, it knows that all the messages, both sent and received through that socket connection have been flushed.

Three things are worth mentioning about the protocol. First, the protocol works even if T1 and T2 are migrated simultaneously. Both tasks will simply go though steps 1, 2, and 3, with the **shutdown()** on step 2 of one task causing an EOF to be read on step 3 of the other task. Second, note that the SIGURG signal handler, just like the migration signal handler, uses pvmlib routines to read the messages from the socket. Thus, the SIGURG signal handler had to guarded against re-entrancy problems, using the same method used for guarding the migration signal handler. Lastly, the protocol described above assumes only one TCP socket connection is being closed. Typically, either Mtask has no TCP socket connections with other tasks or it has a number of them, all of which have to be closed. Instead of executing the protocol one connection at a time, Mtask can execute steps 1 and 2 for all TCP connections first before going to step 3. This causes steps 4, 5, and 6 on all the peer tasks to be executed in parallel, further minimizing the time it takes to take down all the connections.

A drawback of this method, however, is that it doesn't work for Unix domain sockets. PVM 3.3.x uses Unix domain sockets for direct connections between tasks on the same host since it is about 1.5x - 2x faster than TCP sockets [7]. Unix domain sockets only work for tasks on the same host. Unfortunately, Unix domain sockets has no support for OOB data. As currently implemented, MPVM uses TCP sockets for direct communication even for tasks on the same host.

One last aspect related to TCP connection closure is with regards to routing for messages from the peer tasks to Mtask and the re-establishment of the TCP connection after migration. As mentioned previously, TCP connections between tasks are established on demand. Since the peer task has its routing option set to direct routing (which had to be set in the first place for the just-taken-down TCP connection to have been established), the peer task will try to establish another TCP connection on the next message to Mtask. But since Mtask is migrating, this should not happen. Otherwise, the peer task would only be blocked waiting for an acknowledgment from Mtask. To address this problem, before the SIGURG signal handler returns, a flag inside the pvmlib is set to indicate that the peer task should not try to establish a TCP connection with Mtask. Messages for Mtask will then be routed indirectly through the pvmds allowing the peer task to continue executing.

Once Mtask has migrated and is running again, it would be desirable for the TCP connections that were taken down before migration to be re-established. Since a flag has been set on the peer tasks, no TCP connection request will come from the peer tasks. The request should come from Mtask. One option would be for Mtask to request for a TCP connection from all the tasks it used to have a connection with prior to returning from the migration signal handler. However, this option would pay the price of establishing the connection without knowing if the connection will even be used. This brings connection re-establishment back to the "on demand" philosophy.

To continue supporting "on demand" TCP connection establishment, one possibility is to inform all the peer tasks that they could now establish a TCP connection if they wanted to. This option, however, would require that a message be multicast to all the peer tasks. The solution taken in MPVM currently is to do nothing. That is, a TCP connection will be established with a peer task only if Mtask requests for it. This request will be generated on the first message Mtask sends to the peer task after the migration. This implementation, however, implies that if the communication between the peer task and Mtask is always one way from the peer task to Mtask, all the messages will be routed through the pvmds. Both options have advantages and disadvantages. Which one is better is debatable.

3.4 Message Delivery on Migration

An important aspect of the MPVM implementation that has yet to be discussed is how MPVM handles messages for migrating/migrated tasks. That is, how do messages sent to Mtask find their way to the new location of Mtask. To ensure correct delivery of messages in the presence of migration, support for virtual tids, message forwarding, and message sequencing had to be built into MPVM.

Note that the problem of message delivery really only applies to messages using indirect routing. Direct routing is not a problem since by definition, it uses a point-to-point connection. Also, recall that at migration time, direct connections are taken down and messages from other tasks to the migrating task are automatically routed indirectly through the pvmds. The next three sections will therefore be presented in terms of indirectly routed messages.

3.4.1 Virtual Tids

All tasks in PVM are identified by task identifiers or tids. These tids are used to identify the source and destination of messages. Tids are formed using an encoding of a host ID and a task number [8]. The host ID or hid represents the host where the task is executing while the task number identifies a particular task on a particular host.

The combination of the host number and the task number uniquely identifies any task in the entire virtual machine. One advantage of this encoding scheme is that it allows fast routing of messages since the target host of any message can be determined directly from the destination tid. However, recall that the tid of a task is part of the state information maintained on migration. That is, a task with tid T1 will always be referred to as T1 regardless of where it is actually executing. The use of the same tid is necessary to make migrations transparent to the application. Unfortunately, the use of the same tid also implies that there is no longer any guarantee that the host number encoded in the tid is the actual host where the task is executing.

MPVM gets around this problem by virtualizing tids, thus making them location transparent. Virtualizing the tids is done by maintaining a table of tid-to-host mappings. Instead of just relying on the host number encoded on the tid as the search key for the target host, the whole tid is used. Note that the same host number and task number encoding scheme is still used in generating the tids.

Each pvmd in the virtual machine maintains two tid-to-host mapping tables: a *home* map and a *hint* map. The home map on host H, for example, contains a list of mappings for tasks that were originally started on host H, regardless of where they are currently executing. Note that since these tasks were originally started on host H, the host numbers in their tids "point" to host H as their home. The home map on host H is always updated whenever a task whose home is host H migrates.

Consider the example in figure 6. In step 1, task T1 is started in host H1. This causes a T1→H1 entry to be added on the home map of H1. At some later time, step 2, T1 migrates to host H2. This migration causes the T1→H1 home map entry on H1 to be updated to T1→H2, indicating that T1 is now on H2. The same goes for step 3 when T1 migrates to H3. Notice from the figure that when T1 migrated from H2 to H3, a *DM_HOMEUPD* CM was sent from H2, where T1 migrated from, to H1, the home of T1 (step 4). This CM informs H1 that task T1 has migrated to H3, causing H1 to update its home map. It was not necessary to have a *DM_HOMEUPD* CM when T1 first migrated from H1 to H2 since H1 is already the home of T1 and the home map can be updated directly.

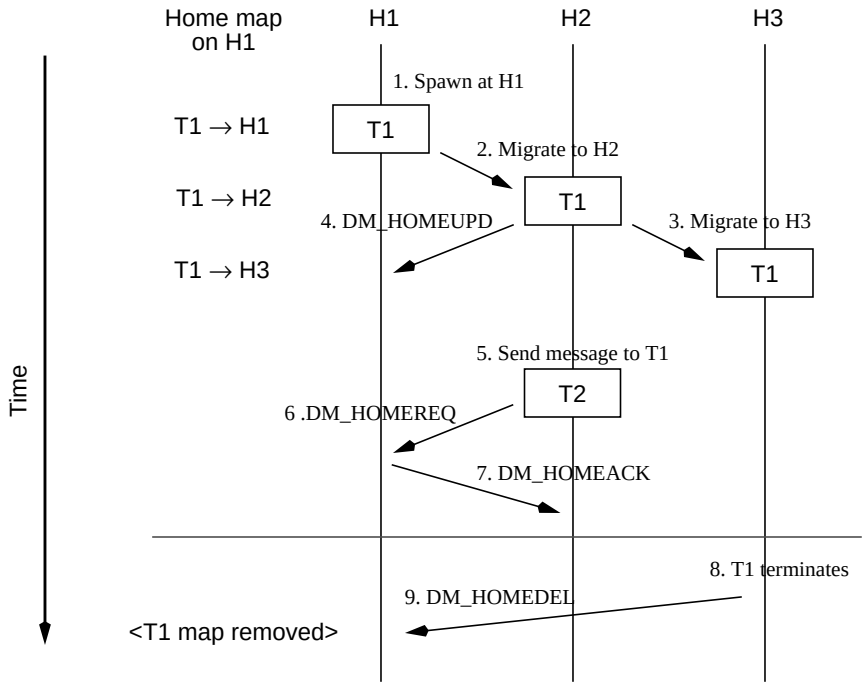

Figure 6. Tid virtualization using home maps. A task's 'home' host is the host where it was originally started and is identified by the host number encoded in the task's tid.

With the use of the home map, it is possible to determine the exact whereabouts of any given task. For example, using figure 6 again, if at some later time task T2 in host H2 sends a message to T1. The message is first routed through the pvmd on H2 (step 5). The pvmd on H2 determines that the destination for the message is T1 and sends a *DM_HOMEREQ* CM to H1, the home host of T1 (step 6). Recall that the home host is determined from the host number encoded in the tid of T1. The pvmd on H1 receives this CM and replies with a *DM_HOMEACK* CM containing the T1→H3 mapping (step 7). The pvmd on H2 then knows that the message should be sent to H3.

When T1 terminates, the pvmd on host H3 sends a *DM_HOMEDEL* CM to the pvmd on H1 indicating that T1 has terminated. This CM causes the home map entry for T1 to be removed. If at some later time a *DM_HOMEREQ* CM for T1 is received by H1, a *DM_HOMEACK* CM containing an error code is sent back to the requesting pvmd. This error code would inform the requesting pvmd that T1 no longer exists and the message for T1 is discarded.

While this scheme works nicely, it is terribly inefficient. To improve the performance, a hint map is used. The hint map caches tid-to-host mappings received from previous home map requests. Using a hint map will minimize the need for sending *DM_HOMEREQ* CMs to only when there is no local copy of the mapping. As currently implemented, the hint map is allocated statically. This limits the number of mappings that could be cached. When the hint map gets full, replacement of entries uses the least recently used (LRU) policy. This policy will throw away the mapping that hasn't been used for the longest time. Examples of such mappings would be those for tasks that have terminated. Note that the hint map could also be updated during task migration. For example, when task T1 was migrated from H2 to H3, it's possible to update the hint map on H2 with the entry T1→H3 immediately. This "aggressive" update of the hint map would eliminate the need for a *DM_HOMEREQ* CM in step 5 of figure 6. In the same manner, hint map entries could be added for newly spawned tasks.

3.4.2 Message Forwarding

With the use of home and hint maps, it is possible to determine the exact location of any task at any time. However, in the face of migration, these home and hint maps

could be left in an inconsistent state. For example, using figure 6 again, the home map in H1 won't reflect the T1→H3 mapping until it receives the *DM_HOMEUPD* CM. If a *DM_HOMEREQ* CM arrived just before the *DM_HOMEUPD* CM, the *DM_HOMEACK* CM reply would contain a T1→H2 mapping which is no longer true. Also, note that no where in the migration protocol are the other pvmds (aside from the source pvmd, target pvmd, and pvmd on the home host) in the virtual machine informed of the migrated tasks' new location. Thus, the hint maps on these "uninformed" pvmds could contain old, and now invalid tid-to-host mappings. The end result of these invalid home and hint maps is that messages will be sent to the wrong host. In this case, the received message should be forwarded to the correct host and the invalid host/hint maps corrected.

Consider the example in figure 7. Assuming H1 is the home host of T1, H1 has a home map entry for T1. In step 1, T1, which is currently in H3 (which means task T1 migrated from H1 to H3, possibly through other hosts) migrates to H4. At almost the same time, T2 on H2 sends a message to T1 (step 2). If H2 had an out-of-date hint map, the message would be sent to H3, the previous host of T1. The pvmd on H3 will determine that the destination task T1 is no longer one of its local tasks. At this point, there are two possibilities: either the pvmd on H3 has an idea of where T1 is (it has a home or hint map entry for T1) or it doesn't.

In the case where the pvmd has an idea of where T1 is, H4 in this case, the pvmd on H3 will send the pvmd that sent the message a *DM_HINTUPD* CM containing a T1→H4 mapping (step 3), and then forward the message to H4 (step 4). The *DM_HINTUPD* CM will cause the pvmd on H2 to update its hint map so that future messages for T1 will be sent directly to H4. Note that the mapping the pvmd on H3 has need not necessarily be valid. Such would be the case if T1 migrated from H4 to some other host again. In that case, the message forwarding sequence will simply repeat.

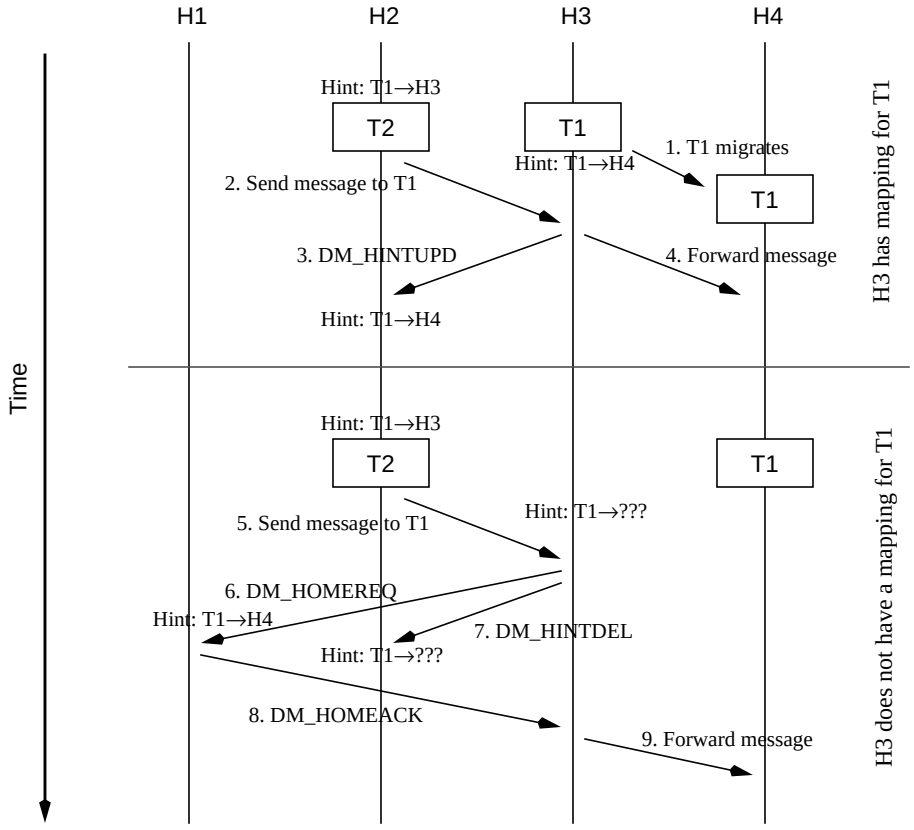

Figure 7. Message forwarding protocol. In the first case, the pvmd on H3 has a mapping (not necessarily valid) for T1. In the second case, the pvmd on H3 has no mapping for T1.

The other case to consider is when the pvmd on H3 has no idea where T1 is currently executing. This case is possible if T1 once migrated to H3, causing a T1→H3 hint map entry to be added on H2, but has since moved to another host and the T1 mapping in H3 has been removed from its hint map due to the LRU policy. Since the pvmd on H3 doesn't know of T1's whereabouts, it sends a *DM_HOMEREQ* CM to H1, the home of T1 (step 6). It then sends a *DM_HINTDEL* CM to H2, the source of the message (step 7). Eventually, the pvmd on H1 will reply with a *DM_HOMEACK* CM to H3 containing the T1→H4 mapping (step 8). H3 updates its hint map and then forwards the message to H4 (step 9).

The *DM_HINTDEL* CM sent to H2 in step 6 causes the incorrect T1→H3 hint map entry on H2 to be removed, forcing H2 to request for T1's location from H1 on the next message to T1. An alternative implementation is for H3 to wait for the *DM_HOMEACK* CM from H1 and send the returned mapping to H2 using a *DM_HINTUPD* CM. This method would update the hint map on H2 eliminating the need for H2 to send a *DM_HOMEREQ* CM to H1 for future messages to T1. The drawback of this method is that while H3 is waiting for the *DM_HOMEACK* CM reply to arrive, the pvmd on H2 may be continuously sending messages to H3, all of which have to be forwarded to H4. By sending the *DM_HINTDEL* CM to H2 immediately, the pvmd on H2 would be forced to get the true location of T1 from H1, allowing the messages to be sent to H4 directly.

3.4.3 Message Sequencing

A consequence of message forwarding, however, is that it could break PVM's message ordering semantics. Consider the situation in figure 8. The situation is similar to the example in figure 7 above except that the message H2 forwarded to H3 (message A in figure 8) takes a long time to get to H3. An example of why message A could be delayed is that H2 on a different network than H1 and H3. Since H1 and H3 are on the same network, a message would travel faster from H1 to H3 than from H2 to H3. The important point here is that the delay, whatever the reason, caused message A to arrive after message B. This behavior is a direct violation of the PVM message passing semantics since message A was sent before message B. It is therefore essential to use some sort of sequencing mechanism to ensure proper ordering of messages.

In standard PVM, the pvmds communicate via UDP sockets for scalability reasons. UDP transport, however, has two basic restrictions. First, a UDP message or datagram can only be UDPMTU (UDP Maximum Transmission Unit) bytes long. The UDPMTU limit is host dependent. This restriction requires messages larger than UDPMTU bytes to be broken up into message fragments or packets. Note that when considering the effective UDPMTU between two hosts, the smaller of the two MTUs is used. For example, the UDPMTU between H1 and H2 is 4096 but only 2048 for H2 and H3. The second restriction is that UDP is unreliable. That is, datagram

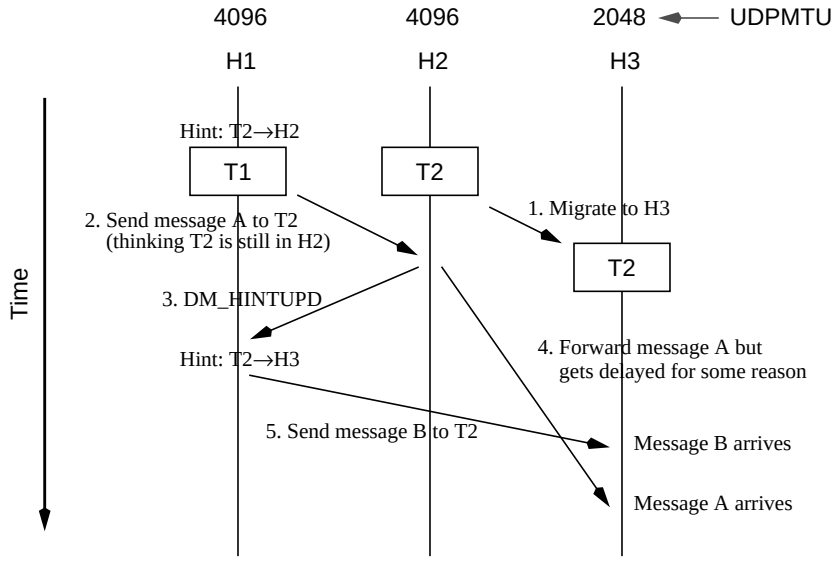

Figure 8. Example where message forwarding breaks message order. In this example, messages A and B are sent from T1 but arrive at T2 in the reverse order, in violation of the PVM message ordering semantics.

delivery is not guaranteed. These two restrictions requires the pvmds to 1) have the ability to fragment and de-fragment messages and 2) support message fragment re-transmission to guarantee delivery.

Taken in this context, the discussion above regarding message forwarding really applies to message fragments rather than whole messages. Thinking of the example in figure 8 in these terms (i.e., messages A and B are actually fragments A and B of one message), it is possible that the whole message arrives but with its contents out of order.

To address this problem, each message sent from T1 to T2 is assigned a sequence number based on the number of bytes already sent from task T1 to T2. That is, the sequence number for message *N+1* is calculated as

$$Seq\#_{N+1} = Seq\#_N + Len_N$$

where $Seq\#_N$ and $Seq\#_{N+1}$ are the sequence numbers of packets *N* and *N+1* respectively, Len_N is the length of packet *N* in bytes, and $Seq\#_0 = 0$ and $Len_0 = 0$.

For example, using the UDPMTU values in figure 8 and assuming T2 is still in H2, if the first message T1 sends to T2 is 6000 bytes long. it will be broken into two fragments with 4096 and 1904 bytes each with sequence numbers 0 and 4096 respectively. A second 6000 byte message from T1 to T2 will again be broken into two fragments of 4096 and 1904 bytes each but will have sequence numbers 6000 and 10096 this time.

Since each message fragment has a unique sequence number, it is now possible to re-arrange the fragments even if they arrive out of order. Note that even if fragments are further fragmented, correct ordering can still be maintained. Fragmentation of message fragments is possible due to different UDPMTUs between hosts. Using figure 8 again, the pvmd on H2 had to forward a message fragment from H1 to H3. If the fragment is 4096 bytes long (UDPMTU between H1 and H2 is 4096), it will have to be further fragmented into two 2048 byte fragments since the UDPMTU between H2 and H3 is only 2048. In the re-fragmentation process, new sequence numbers are calculated, using the same equation above, for each of the fragments. For example, if the original 4096 byte fragment had a sequence number of *S*, after re-fragmentation, the first fragment will have sequence number *S* and the second fragment will have sequence number *(S* + 2048), where 2048 is the length of the first fragment.

Note that message sequence numbers are based on point-to-point messages. That is, the sequence numbers for messages from T1 to T2 are independent of the sequence numbers of messages from T1 to any other task. Since these sequence numbers are based on point-to-point messages, the assignment of sequence numbers and the re-ordering of message fragments based on these sequence numbers is done in the pvmlib. Also note that since the pvmds guarantee delivery of message fragments through re-transmission, the message re-assembly code in the pvmlib, the code responsible for correctly sequencing fragments into messages, need not worry about lost packets.

Another point to mention is with regards to 0-byte messages (i.e., one produced by a **pvm_initsend(); pvm_send()** code sequence). Considering how sequence numbers are calculated, the sequence number of a 0-byte message *N* will be the same as the sequence number of message *N+1*. This situation is obviously unacceptable. Fortunately, what the application sees as a 0-byte message is actually a message with some header information and 0-bytes of application data. The message header contain information such as the message's tag and encoding. Since the message headers are counted as part of the message length, there can never be truly 0-byte messages.

While the sequencing mechanism described above works for point-to-point messages (messages sent by **pvm_send()**), it presents a problem for multicast messages (i.e., messages sent by **pvm_mcast()**).

Figure 9 illustrates the multicast mechanism in PVM as task T1 sends a multicast message to tasks T2 through T5. In step 1, task T1 first sends a list of the target tasks (T2 .. T5) for the multicast message. The pvmd on H1 then determines the hosts where the target tasks are executing. H2 and H3 in this case. The pvmd on H1 then sends a message to H2 and H3 indicating that a multicast message will be sent to all or some of their local tasks (step 2). In the case for H2, the message contains the tids for tasks T2 and T3 indicating that the multicast message will be for tasks T2 and T3 only (assuming there are other tasks on host2). The same goes for H3. Note that no message is sent to H4 since T6 is not a recipient of the multicast message. In step 3, task T1 sends the actual message. A copy of the message is then sent by the pvmd on H1 to H2 and H3 (step 4). The pvmds on H2 and H3, knowing which local tasks the multicast message is meant for from step 2, send each of the target tasks a copy of the message.

The advantage of this implementation is that regardless of the number of target tasks on H2 for example, only one message will be sent from H1 to H2. The pvmd on the target host is responsible for giving each target task a copy of

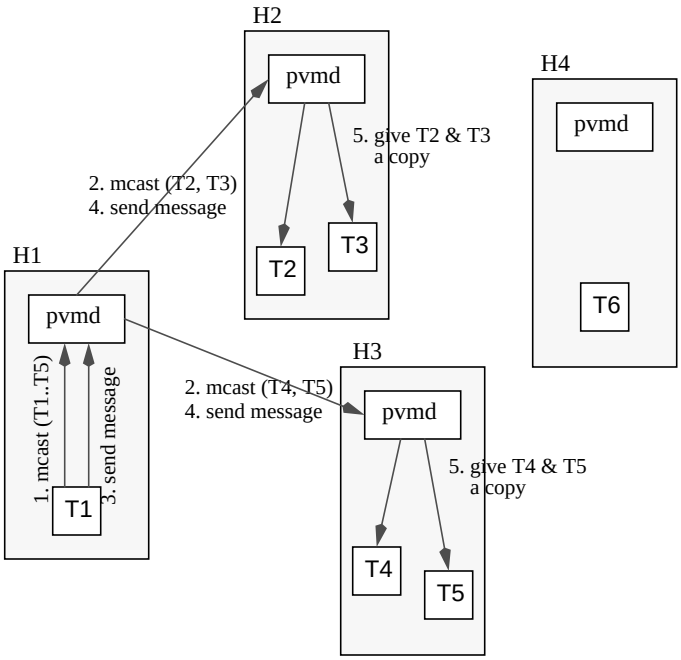

Figure 9. Mutilcast mechanism in PVM. This example shows the steps involved in sending a multicast message from T1 to tasks T2 to T5.

the message. Unfortunately, the fact that only one message is sent out by T1 causes some problem with the sequencing mechanism discussed above. Conceptually, a multicast message to N tasks is functionally equivalent to a point-to-point message to each of the N tasks. Since the sequence numbering is based on the number of bytes previously sent from the sending tasks to the target task, each of these N point-to-point messages will most likely have different sequence numbers. Considering only one message is sent from the sending task for a multicast message, there is a problem on how each of the N target tasks will receive the message with the appropriate sequence number.

This problem is resolved by sending the “would be” sequence number as part of the list of target tasks. Going back to the example in figure 9, assume that the correct sequence number of the next message for each for the four target tasks T2 to T5 are 200, 300, 400, and 500 respectively. In step 1, instead of sending just the list of target tasks, the list of <tid, sequence number> pairs is sent. The same approach is used in step 2 where instead of just sending a message containing the tids of T2 and T3 to H2, a message containing <T2, 200> and <T3, 300> is sent. When the actual message is sent (steps 3 and 4), the sequence number of the message is reset to 0. That is, if the message is 6000 bytes long and the UDPMTU is 4096, the resulting two message fragments would have sequence numbers 0 and 4096. Finally on step 5, as the pvmd on H2 gives task T2 a copy of each fragment of the message, it adds the sequence number for T2 it got from step 2 to the current sequence number in the fragment. By updating the sequence number, task T2 will receive two message fragments with sequence numbers 200 (200 + 0) and 4296 (200 + 4096), which are the sequence numbers of the next message T2 expects from T1. Task T3 will receive two fragments with sequence numbers 300 and 4396 and so on for tasks T4 and T5. Note that because the sequence number for the message fragments is reset to 0, this mechanism will work even if the multicast message fragments get further fragment along the way.

Aside for the modifications described above, more changes had to be made for the multicast mechanism to work under MPVM. Recall that the pvmd on H1 first had to determine the hosts on which the target tasks were executing before it could go to step 2. For this purpose, the home and hint maps are used. It is possible, however, that some of the target tasks don’t have an entry on the home or hint maps. One solution is to send a *DM_HOMEREQ* CM for each tasks that doesn’t have an entry, and only go to step 2 when all the corresponding *DM_HOMEACK* CMs are received. While this solution would work, it has the tendency to delay messages unnecessarily. For example, if

the pvmd on H1 had a mapping for all but T5, tasks T2, T3 and T4 will have to wait until the pvmd on H1 gets the *DM_HOMEACK* CM for T5.

To avoid unnecessarily delaying the message for all the tasks, instead of sending a *DM_HOMEREQ* CM for T5, the pvmd on H1 will assume that T5 is in its home host, H3 in this case. If T5 was actually on H3, then everything is fine. If it so happened that T5 was actually on H4, in step 5, the pvmd on H3 will still update the sequence numbers for T5's copy of the message, but will recognize that T5 is not executing locally. This will cause the pvmd on H3 to execute the message forwarding mechanism described previously on the copy of the message for T5. Since H3 is the home node of T5, it knows exactly where the message should be forwarded from its home map. Also, recall that the message forwarding mechanism sends a *DM_HINTUPD* CM to the source of the message, H1 in this case. This CM will cause the pvmd on H1 to update its hint map so it knows where T5 is located on the next message send or multicast involving T5. This is the same sequence that would happen if the home or hint map on the source of the multicast address had an out-of-date entry for some of the tasks. Such would be the case if the pvmd on H1 had an incorrect T2$\longrightarrow$H3 mapping. The only difference being that a *DM_HOMEREQ* CM might be generated if H3 has no idea where T2 is located.

The use of sequence numbers for multicast messages raises an issue with regards to the message ordering semantics defined by PVM. As mentioned earlier, PVM requires that messages from task A to task B should be received in the same order they were sent. However, this condition actually breaks in PVM when messages are sent using both **pvm_mcast()** and **pvm_send()** with direct routing. Consider the case when a message is sent from task A to task B using **pvm_mcast()** followed by another message sent via **pvm_send()** with direct routing. Since **pvm_mcast()** routes messages through the pvmds while the **pvm_send()** uses a direct TCP connection with task B, it is more than likely that the message sent via **pvm_send()** will get to task B first, in violation of the message ordering semantics. With the use of sequence numbers for both point-to-point and multicast messages, the message ordering semantics can be preserved. Whether this property gives MPVM some advantage over PVM is hard to say. However, at the very least, mixing **pvm_mcast()** and **pvm_send()** with direct routing will now generate deterministic application behavior.

3.5 Migrating OS state

OS held state cannot be transferred like the processor state, the process' data or stack. For one thing, since the migration mechanism is implemented at user level, not all OS held state can be captured/reconstructed. An example of process state that cannot be reconstructed is the process ID. Recall that PVM tasks are actually Unix processes. As such, they have assigned process IDs. Allocation and assignment of process IDs to processes is done entirely by the OS kernel.

Realize that it is only necessary to migrate OS state information that the process can observe directly. For example, OS kernels keep track of the page table entries of processes. But since processes are "usually" not concerned about the specifics of these page table entries, on migration, the OS kernel on the target machine could be left alone in deciding how to allocate pages and page table entries.

The problem in migrating OS state is that the OS state a process observes in valid only in the context of the computing environment at the time the state was observed. Changing the computing environment (e.g., the process migrates from one host to another) would require a mapping of the OS state information as viewed by the process to its equivalent in the new computing environment. This mapping or virtualization of OS state can be achieved to some extent by providing "wrappers" to system calls.

Consider the case of file I/O. To accommodate file I/O migration, the pvmlib supplies its own file I/O routines (e.g., **open()**, **close()**, **dup()**, etc.) which are wrappers for the actual system calls. These wrapper functions allows the pvmlib to maintain a list of the files used by a task. This list contain information such as the file's name, file access mode, file descriptor, etc. On migration, but prior to the actual state transfer, for each file in the used files list, the current file pointer offset is taken and then closed. Upon restart, each file in the list is re-opened and the current file pointer is reset to its position prior to migration. The pvmlib also makes sure that each re-opened file is assigned the same file descriptor used before migration.

For file I/O migration to work, as currently implemented, it is necessary that the file be available on the target host. For simplicity, a global file system is assumed to exist (e.g., through NFS). Ways of getting around this restriction are currently being investigated. The current MPVM pvmlib traps only commonly used file I/O system calls such as **open()** and **close()**. There is currently no support for **fcntl()** and **ioctl()**, for example.

4 Quantitative Evaluation

This section presents performance results for MPVM. The first two experiments were designed to measure the normal case performance (i.e., no migration) of MPVM against PVM 3.3.4 at the micro-benchmark and application level. The third experiment was designed to test migration performance. All experiments that required timing measurements were done on two idle HP series 9000/720 workstations running HP-UX 9.03 connected over an idle 10 Mb/sec Ethernet. Each workstation has a PA-RISC 1.1 processor and 64 MB main memory.

4.1 Ping

Table 1 shows the results of running a "ping" experiment using PVM and MPVM. The ping experiment was set-up to determine the difference between the message passing times of the two systems. In this experiment, a message is sent from one host to another and back. There is very little computation done. To take the steady state performance for each data size, 50 messages of the appropriate size are first sent back and forth to "warm-up" the system. After which, a timed execution of 1000 ping messages was done.

As can be seen from table 1, MPVM and PVM only differ in the order of 10ths of a millisecond. In general, however, MPVM is expected to be slower than PVM for three reasons. First, there is the additional cost of avoiding potential re-entrancy problems. Every time task execution enters/leaves the pvmlib, a flag has to be set/reset. Second, there is the cost of virtualizing tids. This cost only applies for indirectly routed messages. For every message sent out through the daemon, a table lookup has to be done to determine the correct location of the target task. The cost is even greater the first time a lookup is done since a first time lookup would typically result in a *DM_HOMEREQ* CM mapping request from the home node of the destination task. Lastly, there is the cost of supporting sequence numbers. This sequence numbering cost is linear with the number of fragments of a message.

Notice that even as the message size increases, the percentage difference between the performance of PVM and MPVM decreases despite the fact that the cost of sequence numbering is linear with the message size (i.e., the larger the message, the more it will be fragmented). This result indicates that as message size increases, the cost data transfer increasingly dominates the cost of sending a message.

It should also be mentioned that in the case of direct connections between tasks in the same host, the performance of PVM is better than that of MPVM. This performance difference is due to PVM's use of Unix domain sockets for direct connections between tasks in the same host. MPVM does not use Unix domain sockets because it doesn't support OOB data which is used for asynchronous closure of direct connections. Ways of getting around this restriction are currently being investigated.

4.2 Gaussian Elimination

The Ping experiment above showed the overhead MPVM imposes on message passing performance. Though good for benchmarking, it can hardly qualify as a real-world application since barely any computation was done. To show how MPVM affects the performance of real-applications, a parallel Gaussian elimination program was run using both PVM and MPVM for different matrix sizes. This experiment only used two machines, each task

Data size	PVM (ms)		MPVM (ms)	
	Indirect	Direct	Indirect	Direct
0	4.788	1.693	4.991	1.990
1024	7.566	4.035	7.760	4.321
2048	9.533	5.432	9.612	5.648
4096	17.751	10.059	18.015	10.493
8192	29.990	18.849	30.024	18.221
16384	53.370	33.694	53.353	33.871
32768	102.478	65.211	102.187	65.574

Table 1: Ping experiment results for PVM and MPVM for both direct and indirect communication modes. The numbers represent the average roundtrip time of a message.

being responsible for solving half of the matrix. As can be seen from Table 2, the overhead imposed by MPVM is hardly noticeable.

Matrix size	PVM (sec)	MPVM (sec)
80 x 80	0.448	0.453
300 x 300	3.205	3.205
500 x 500	9.311	9.375

Table 2: Gaussian elimination timing results for PVM and MPVM.

4.3 Migration Cost

In this section, the cost of migrating a task is presented. Two measures are defined. The first is the obtrusiveness cost, i.e., the time from when the Spvmd receives an *SM_MIG* CM to the time the task is removed from the machine (i.e., the task exits). This time represents the minimum time an owner may have to wait before regaining dedicated access to the machine. Note that the machine is not necessarily unusable during this time; it just means that something else is executing other than the owners jobs. The second measure is the migration cost. This is the time from when the Spvmd receives the *SM_MIG* CM to the time the task has restarted on the destination host. The migration cost is essentially the obtrusiveness cost plus the restart stage cost. The first measure approximates the impact of migration on the owner, the other on the job itself. Table 3 and figure 10 show the obtrusiveness and migration costs for migrating the Gaussian elimination program used in the previous section for various matrix sizes.

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 3: 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.

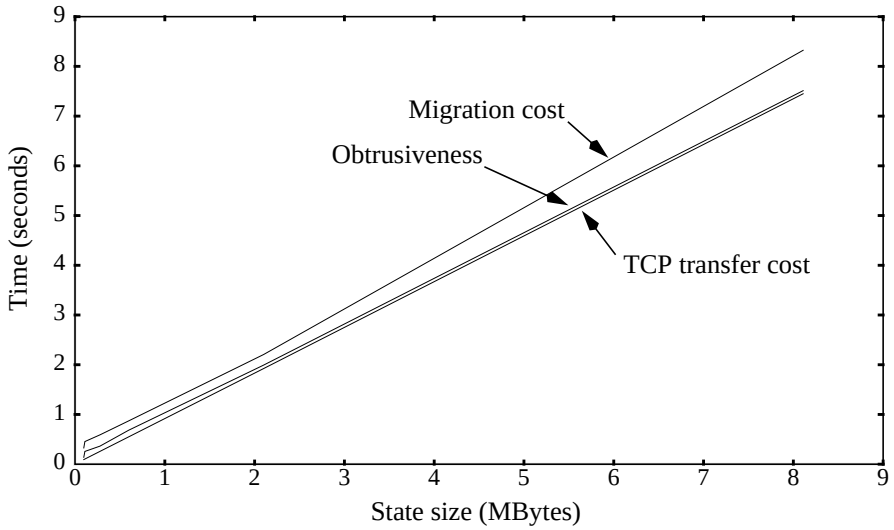

Figure 10. Graph of Table 3. This graph more clearly shows the direct relationship between the state size, the TCP transfer cost, and the migration cost.

As in the previous experiment, this experiment makes use of two machines with one task on each machine. The timing measurements were taken while migrating one task on one machine to the other. The “process state size” indicates the actual number of bytes transferred as measured at migration time. This state size includes static and dynamic data and the stack. The “TCP transfer time” shows the time spent just transferring the same amount of data over a TCP socket connection. This measure provides a lower bound for the migration cost. As can be seen from the table, the task’s state size is the dominating factor in the obtrusiveness and the migration cost

While the effect of the migration on the migrating task can be quantified in terms of the migration cost, the effect of migrating one task on the whole application cannot be as easily defined. In the best case, the migration of one task may not affect the performance of the application at all if, for example, the migrated task had the least work to do or was blocked waiting for the other tasks anyway. In the worst case, the entire application could be stalled by as much as the migration cost if, for example, migration occurred just before a global synchronization point (e.g., a barrier), effectively stalling all the tasks in the application until the migrated task resumes execution.

A tightly-coupled application is likely to be affected more than a loosely-coupled one. Take the case of a 5-node Gaussian elimination application (figure 11). This application is considered to be tightly-coupled due to the large number of messages sent between its tasks. The CPU% axis represents the average % of CPU used by each task on each host since the application started.

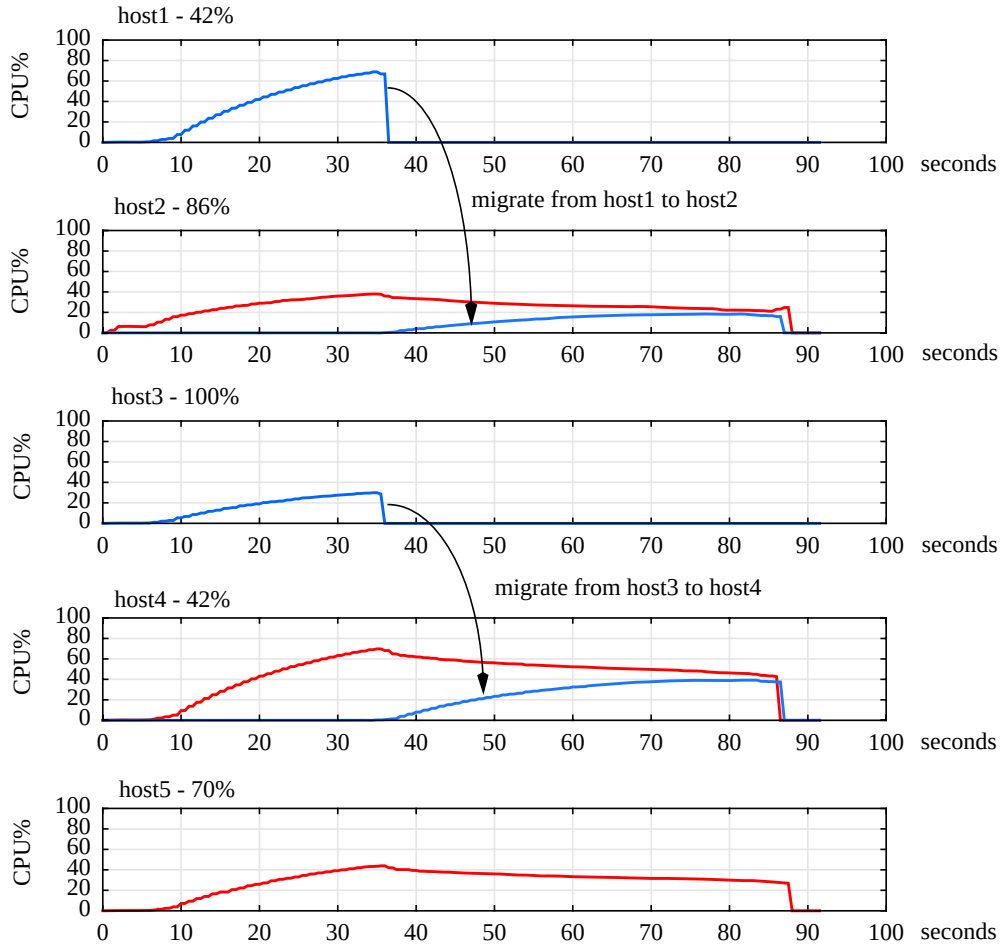

Figure 11. A 5-node parallel Gaussian elimination application. This figure shows the effect of migration on a tightly-coupled application. The CPU% represents the average % of CPU used by the application since it started. CPU% readings were taken every 0.5 second of the application run. The numbers beside the host names (upper left-hand corner) represent the speed index of each machine for this application.

The machines used in this experiments have varied processing speed. The speed index of each machine is shown beside the host name in the figure. The speed index was taken using a 1-node version of the Gaussian elimination program on each of the machines. This difference in processing speed accounts for the difference in CPU% usage on the different hosts. This result is due to the tight-coupling where the tasks on the faster machines are being limited by the task on the slowest machine. As expected, the CPU% of the tasks originally running on both *host2* and *host4* decreased after the migrations. This result is due to the processor sharing that occurred when the hosts were loaded with two tasks. Notice, however, that the CPU% of the task on *host5* also decreased even though it was not involved in the migration. This again can be attributed to the tight-coupling of the application. In this particular case, *host4* being the slower of the two doubly-loaded machines, becomes the bottleneck.

Compare the results in figure 11 with a similar experiment shown in figure 12. Instead of using the parallel Gaussian elimination program, a parallel TSP solver is used. Note that the speed index for this application is different from that of the parallel Gaussian elimination program. The parallel TSP solver is a loosely-coupled application where each task does its own local search and only occasionally exchanges information with the other tasks. The first thing to notice is that the CPU% of all the tasks rise at the same rate on all the hosts. This is due to the loose-coupling of the application, allowing each task to continue executing without being limited by the other tasks regardless of whether they are executing on a fast or

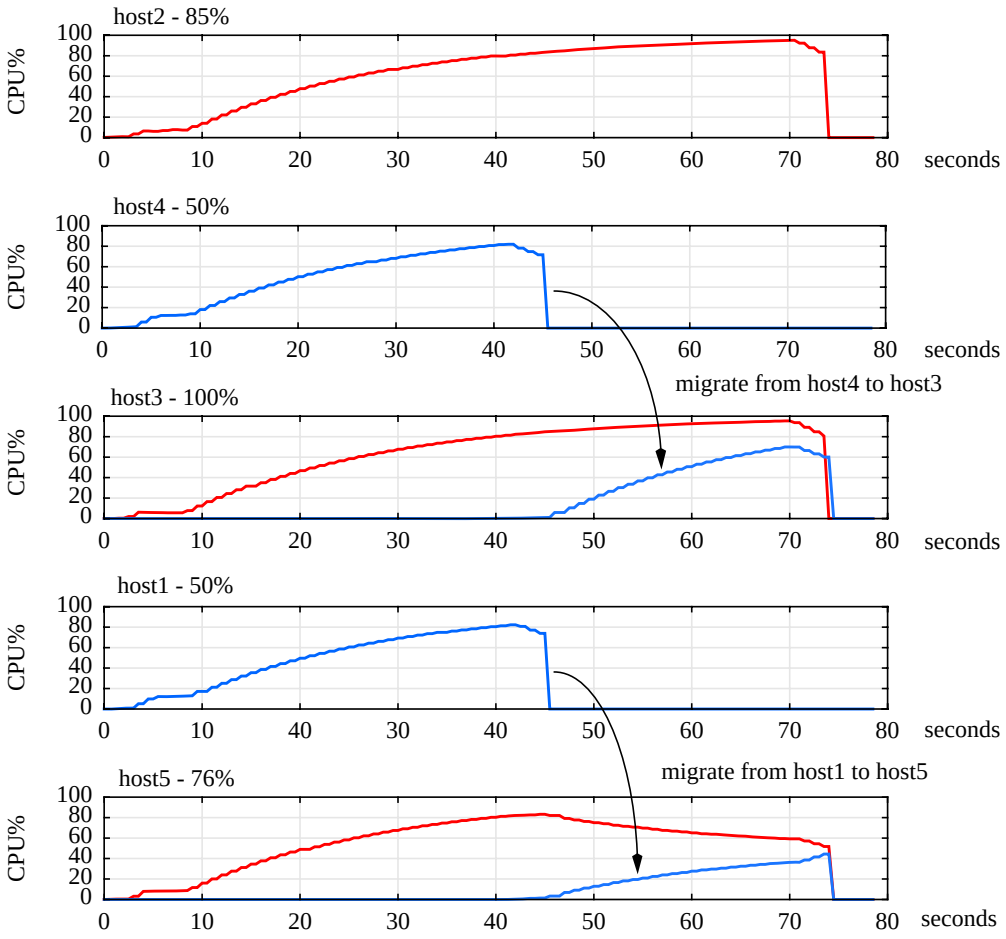

Figure 12. 5-node parallel TSP solver. This figure shows the effect of migrating a task of a loosely-coupled application. Note that *host3* is actually a 2 processor machine which is why the CPU% of the original task running on it is not affected by the migrated task unlike in *host5*. Also note that despite the over-loading of processors (*host5*), the CPU% of the task on *host2* continues to increase

slow machine. The second thing to notice is that the CPU% of the task originally running on *host3* is not affected by the task that just migrated to it. This is because *host3* is actually a two processor machine, and in this case, each processor runs one task. Compare this result with that of *host5* where both tasks are sharing one processor. As expected, the CPU% of the original task would decrease due to the sharing. The last thing to notice in this figure is that unlike in the parallel Gaussian elimination program where the CPU% of the task that was executing all by itself decreased (in *host5*), the CPU% of the equivalent task in the parallel TSP solver, the task on *host2*, didn't decrease. In fact, the CPU% continued to increase, indicating that it is not affected by the doubling-up of tasks on *host5*. This result can be attributed to the loose-coupling of the application.

The significant thing these two experiments show is that the migration protocol in MPVM, by itself, does not impact the performance of tasks on non-involved nodes. The experiments also shows that task placement is another important parameter to be considered when trying to evaluate the effect of migration on the whole application. This observation, however, is much more directed towards scheduling policies, rather than the migration mechanism itself.

5 Related Work

Process migration implementations can be broadly categorized as either supported at the system-level or at the user-level. In system-level supported implementations, the OS kernel is involved in the migration. Notable examples of such implementation are present in Charlotte [9], V [10], Mosix [11, 12], Sprite [13, 14], and Mach [15]. User-level supported process migration implementations, on the other hand, do not require services other than what the OS ordinarily provides through its system call interface. Condor [4, 6, 16] is an example of such an implementation. MPVM fits into this category.

Most of the literature on these systems focus on process migration mechanism efficiency. Efficiency is defined in terms of application "freeze time" and state transfer cost. Freeze time is the time during which the process is not executing. State transfer cost on the other hand is the cost of transferring the process' state. The state transfer cost is often the limiting factor in migration speed [17]. This observation is in accordance with the experimental results shown in the previous section. Two other categorizations of these implementations are in terms of transparency and residual dependency. Transparency refers to how much, or how little, the process is affected by the migration. Of particular interest is how IPC connections are maintained with other processes. Residual dependency refers to how much or how little a migrated process depends on its previous host or hosts.

In Charlotte, the entire virtual address space of a process is transferred at migration time. MPVM uses the same approach. While this approach is simple, it has drawbacks. First, the process is "frozen" for the entire duration of the transfer. Second, the entire virtual address space is transferred even if not all pages may be used by the process. For IPC connections, message senders are informed of the new location of the migrated process. No explicit message flushing is necessary due to kernel provided message caching and retransmission mechanisms. The V kernel addresses the problem of prolonged freeze time by using a technique called "pre-copying". In essence, while the virtual address space is being transferred, the process is allowed to continue executing. Once the transfer is complete, the process is stopped, and the memory pages that were touched by the process after the first transfer started are re-copied. The second-stage transfer hopefully is much shorter thus minimizing the freeze time of the process. This technique has been shown to reduce the freeze time significantly though it has to do more work since some pages have to be copied more than once.

While MPVM could benefit well from pre-copying this technique requires access to page table entries - a requirement that cannot be easily satisfied at user-level. As for IPC, the V kernel allows messages to be dropped, while the process is frozen. Once the process is un-frozen, the senders are informed of its new location and must re-send the messages. MPVM differs in this regard since MPVM doesn't drop messages. Rather, messages are forwarded to the new destination of a migrating task. If the message arrives at the destination and the migrating task hasn't restarted yet, the pvmds simply buffer these messages until the migrating task is ready to receive them.

Sprite takes a different approach by making use of a network-wide file system. Since Sprite uses the network file system as a backing store for virtual memory, most of the memory pages used by a process are already saved on the network file system. Hence, at migration time, all Sprite has to do is to flush all the dirty pages of the migrating process and start-up a process on the destination host whose pages are set-up to be demand-paged from the network file system. Mosix uses a slightly different approach for minimizing freeze time by sending all the dirtied pages of the migrating process directly to the target host but sets-

up the other pages to be demand paged from the executable file (e.g., the text) or zero-filled. Again, while the techniques used by Sprite and Mosix to minimize freeze time could be used in MPVM, these techniques require access to not only the OS' page table entries, but also the systems swap area (backing store). As for IPC migration transparency, Sprite, Mosix and MPVM use different approaches. Sprite uses the shared file system as the medium for inter-process communication. The file system essentially provides a "well-known" point of communication regardless of where a process is actually executing. In Mosix, transparency is easily achieved since most of the process state information is location independent by design. The obvious disadvantage of MPVM over these two systems is that MPVM since MPVM is implemented at user-level, its migration mechanism has to contend with the peculiarities of the OS it is running on as opposed to re-designing the OS to easily accommodate process migration.

Mach uses the concept of Copy-On-Reference (COR) initially used in Accent [17], the precursor of Mach. COR semantics allows a migrating process or task (in Mach parlance) to be started immediately at the target node. When the new task references a page that hasn't been transferred, a page fault occurs and the page fault handler arranges for the page to be sent from the source host (i.e., demand paging from the source's memory). Compared to MPVM's migration mechanism, this method has the advantage of minimal freeze time and minimal state transfer cost since only those pages actually used by the task are transferred. However, it suffers from residual dependency, due to the fact that resources on the source host cannot be released until either everything has been sent or the task terminates. This residual dependency also makes the process susceptible to failure since if any of the hosts on which the process depends on fails, the process could also fail. While MPVM's migration performance may be dwarfed by Mach's use of COR, MPVM doesn't suffer from residual dependencies. To address the transparency problem for IPC connections, a version of Mach that provides in-kernel IPC and DSM called Mach NORMA was used. This method of addressing the IPC transparency issue is very similar to that of MPVM since both systems provide a layer of communication end-point virtualization: the in-kernel IPC in Mach and the message forwarding and task-to-host mappings in MPVM.

On the other side of the implementation domain are those systems implemented at user-level, just like MPVM. The most notable of user-level process migration implementations is Condor. Condor was initially designed for sequential programs. Recently, however, support for PVM applications was added but only as far as scheduling and process suspension/resumption [18]. There is currently no support of migration of PVM applications. The main difference between Condor and MPVM is that Condor uses a checkpoint/roll-back mechanism to achieve migration. This design decision was made to minimize obtrusiveness. The Condor system, from time to time, takes a snap shot of the state of the programs it is running. This is done by taking a core dump of the process and merging it with the executable file of the process to produce a checkpoint file. At migration time, the currently running process is immediately terminated. It is later resumed on another host, based on the latest checkpoint file. In addition to being minimally obtrusive, this method has the advantage of fault-tolerance in that if something goes wrong (e.g., the system crashes), it is still possible to restart the program from the last checkpoint file. Fault-tolerance is something MPVM currently doesn't support. Restarting processes based on roll-backs, however, requires idempotent file operations, a problem MPVM does not suffer from since the state is restarted exactly at the point where it was interrupted. Using roll-backs is particularly troublesome for a parallel/distributed application since it would require synchronous checkpointing of all the tasks or some form of message logging mechanism. MPVM also has the additional advantage of requiring almost no disk I/O. Disk I/O will only occur when transferring pages of the migrating task that have been paged out. Aside from the speed factor, disk space consumption is also avoided. It is not uncommon to see core dumps in the megabyte range.

Two systems closely related to MPVM are UPVM and DynamicPVM. UPVM [19], another research effort here at OGI, addresses the problem of the course-grained distribution granularity present in MPVM. MPVM migrates tasks at the level of whole processes. UPVM introduces the concept of User Level Processes (ULPs) which are thread-like entities that are independently migratable. Since ULPs are smaller "processing" entities than processes, UPVM has the potential for achieving better load balance. As currently implemented, UPVM has two main restrictions. First, it only runs SPMD programs. Second, since all the ULPs share the address space of a single Unix process, there is a limit on the number of ULPs the application can have depending on the size of the virtual address space of the process and the memory requirements of each ULP. DynamicPVM [20] is an extension to PVM to support process migration, very much like MPVM, but relies on Condor-style checkpointing. A more detailed comparison of both systems is unfortunately unavailable at this time.

6 Discussion and alternative implementations

In this section, more qualitative aspects of the design of MPVM, its problems and alternative implementations are discussed.

6.1 PVM Source Code Compatibility

Recall that one of the goals of MPVM is to be source code compatible with PVM. To this end, MPVM has maintained the same user-interface, their parameters and semantics, as defined by PVM. The **pvm_sendsig()** routine has to be specially mentioned however. Since MPVM does not currently support migration of user-installed signal handlers, the use of **pvm_sendsig()** may behave differently for a migrated task that uses signal handlers.

A closely related aspect that affects source code compatibility is the use of a GS. When a GS is used, some of the PVM user-interface calls, **pvm_spawn()** for example, are forwarded the GS. To maintain full compatibility, the GS should respond to these requests in a PVM compatible way. What the GS does with the requests it receives is outside the control of MPVM.

6.2 Portability

Another goal of MPVM is that of portability. This was the motivating factor for choosing a user-level implementation. MPVM was first implemented on HP-PA workstations running HP-UX 9.03. It has since been ported onto SunOS 4.1.3, DEC OSF/1 V1.3, and AIX 3 rel 2.

Although machine dependence of the migration mechanism is unavoidable, the dependence was limited by implementing the migration mechanism using signals, sockets, the **setjmp()/longjmp()** function, etc., all of which are available on most Unix flavors. Also, no assembly language was used. Everything is written using "C" code.

As long as a process can determine the extents of its data and stack segments at run-time, porting the migration code should not be difficult. Consider the difference between the HP-UX and SunOS versions of MPVM for example. For HP-UX, the following macros are defined

```
#define STACK_TOP       ((char *) &stk_var)
#define STACK_BASE      ((char *) USRSTACK)
#define DATA_TOP        ((char *) sbrk (0))
#define DATA_BASE       ((char *) &__data_start)
```

For SunOS, the same macros are defined as

```
#define STACK_TOP       ((char *) USRSTACK)
#define STACK_BASE      ((char *) &stk_var)
#define DATA_TOP        ((char *) sbrk (0))
#define DATA_BASE       ((char *) &environ)
```

USRSTACK is a system defined macro which is the absolute address of the beginning of the stack. *Stk_var* is a local variable defined in a function where these macro definitions are used. *&stk_var* thus provides the process an approximate top of stack address which is always more than what is needed to restore, but only as much as the amount of stack space used by a stack frame on a function call. The *__data_start* and *environ* variables define the start of the data space under HP-UX and SunOS respectively. The *__data_start* variable is documented in HP-UX. The *environ* variable on the other hand is not documented but could be determined by using the **nm** Unix command. And lastly, **sbrk()** is a system call, which when given the parameter 0, returns the address of the top of the heap. Thus, when porting to a new system, only the equivalents of these four definitions need to be determined. In most machines, the usage of **sbrk()** and *&stk_var* should be portable and since *USRSTACK* is usually defined by the system, this leaves only the value of DATA_BASE to be determined.

Unfortunately, there are some systems that don't have the USRSTACK macro defined. In this case, the easiest thing to do is to let the process figure out the start-of-stack address at run-time. One way of doing it is to get the address of a local variable declared in the pvmlib's **main()** and "round" that address to the next higher or lower page boundary. Rounding up or down of the address depends on whether the stack grows downward or upward respectively. The resulting address is the start-of-stack address. While this work around is totally portable, it would fail if the local variable was not allocated on the first stack page. This situation is possible, for example, if enough command line arguments to fill-up the first page of the stack are passed to the process.

Other potential problems are usually caused by system interface incompatibility. For example, some systems use the **sigvec()** interface to install signal handlers while others use **sigvector()**.

There are special cases however that would require more in-depth investigation. For example, the HP-PA workstations use space registers that contain the addresses of a process' text, data, and stack spaces which are guaranteed to be constant for the lifetime of the process. With migration however, these addresses are bound to change,

and would have to be explicitly updated to the new addresses. Fortunately, the signalling facility in HP-UX (as well as in other OSs) provides a third parameter to the signal handler called the signal context. This signal context contains the processor state that was saved when the signal was invoked. Using the signal context, the values of space registers can be updated before returning from the signal handler. Another example of this special case is how well the **longjmp()** code interacts with the use of signal handlers that use a temporary stack. Such was the case in the OSF/1 V1.3 port. The **longjmp()** code had safety-checks that detected an error when used with a temporary stack when in fact there is none. Fortunately, the system also provides a lower level **_longjmp()** function that is essentially a **longjmp()** without the error checking.

6.3 Transparency

The decision to implement migration at user-level for the sake of portability unfortunately had a negative impact on MPVM's capacity to be truly migration transparent. MPVM can only guarantee transparency for PVM interface calls and some file I/O system calls. Again, there is the assumption that a global-file system is used.

By implementing the migration at user-level, state information managed by the OS kernel such as process IDs and pending signals cannot be automatically preserved on migration. Additional transparency problems appear if the task directly uses Unix facilities that depend on the location of the task. Examples of such facilities are shared memory, pipes, semaphores, sockets, and shared libraries.

When developing applications of MPVM, special attention has to be given to shared libraries since most compilers/linkers/bundled libraries nowadays are configured to use shared libraries when available. The developer should explicitly create executable files that are statically linked. This requirement is usually satisfied through some compiler or linker option. Also, recall that the *-Dmain=Main* C compiler flag should also be set.

A possible solution to address this transparency issue in user-level implementations is to provide wrapper functions just like those used for file I/O in section 3.5 for all system calls. These wrapper functions would serve as a layer of indirection between the process and OS effectively virtualizing the state of the OS as viewed by the process.

6.4 Heterogeneity support

MPVM supports heterogeneity at the same level as PVM in that processes can be started-up on both homogeneous and heterogeneous architectures. However, migration can only occur within homogeneous machine pools. For example, given ten machines (five Suns and five HPs), a task can be started on each machine. A task on an HP machine however can only migrate to any of the four other HP machines.

The difficulty in supporting heterogenous migration is that process state on heterogenous machines is represented differently. Heterogeneity can come in the form of different processors and instruction sets, different OSs, different memory management units, etc. Translation of a process' state as captured on one machine to one of a different architecture is not easy, though there is some work being done that addresses this problem [21].

6.5 Scalability

Recall that the migration protocol generally only involves the migrating task, the source pvmd, the destination pvmd, and the home pvmd of the migrating task. This approach implies that regardless of the number of tasks on the system, the operations required to migrate a task remain the same. The involvement of other tasks would only depend on whether they have TCP connections with the migrating task that have to be closed. Other than that, all other tasks will continue executing as they normally would and will only get affected by migration if they require a message from the migrating task (i.e., application level synchronization). Also, note that the migration of one task is totally independent of the migration of another. This "independence" property of the migration protocol allows multiple simultaneous migrations to occur. For these reasons, the migration protocol in MPVM could claim to be scalable in 1) the number of nodes, and 2) the instability of the environment.

A factor that negatively affects the migration protocol's scalability, however, is the assumption of the existence of a global file system. MPVM currently relies on a global file system in two ways. First, to avoid moving the migrating process' text, it assumes the executable file of the migrating process is available on the destination machine. Second, the current support for migration transparent file I/O assumes that files available on the source machine are also available on the destination machine. Both of these assumptions will only be always true under a global file system. While such global file systems already exist, the reality is that such file systems are not yet com-

mon place, though it is certainly possible to "simulate" one via NFS, for example.

6.6 Performance

As has been concluded by other studies and also from the task migration cost measurements in section 4, the performance of the migration mechanism is largely dependent on the cost of transferring the process' virtual address space. An optimization already done was not to use checkpoint-style migration. The initial implementation of MPVM used a Condor-style mechanism where core dumps were taken and a checkpoint file was generated by merging data from the core file and original executable file. By moving to direct state transfer through a TCP connection, however, the migration speed was increased approximately 10x for processes which use lots of memory.

The current implementation of MPVM is very much the same as that of Charlotte. That is, the entire virtual address space (data and stack at least) is transferred at migration time. As mentioned above, this has two drawbacks. The first is prolonged freeze time and the second is possible waste of work by transferring all the pages in the virtual address space even though not all may be used.

Unfortunately, current OSs don't leave much of a choice as far as user-level implementations are concerned. The solutions presented by systems such as V and Mach rely on virtual memory functions such as trapping page faults, checking for dirty pages, etc. These functions, however, are not generally available at user-level. Though there is work being done to provide user-level virtual memory management [22, 23, 24], until such functionality becomes widely available, portable user-level process migration implementations cannot make use of methods available to system-level implementations

7 Conclusion

MPVM is an extension to PVM that provides for transparent process migration. Such a facility allows tasks to be scheduled on a machine and then later moved to another if so desired. This ability to move tasks makes it possible to use idle cycles on available machines and at the same time respect ownership of those machines. As is, existing PVM applications can be used under MPVM with little modification. Migration is transparent to the application developer as far as the PVM interface is concerned. File I/O migration is also supported to some extent. Versions of MPVM currently exist for HP-UX 9.03, SunOS 4.1.3, DEC OSF/1 V1.3, and AIX 3 rel 2. Micro-benchmarks show that message-passing in MPVM is just slightly slower than that of PVM. However, tests with real-world applications such as the Gaussian elimination program where some amount of computation being done show that this difference in latency is barely noticeable.

To ensure that task migration doesn't affect the correctness of the application, a strict migration protocol is used. The protocol ensures that messages are not lost and are received in the correct order. The design of the protocol is scalable such that the migration of a task is not affected by the number of tasks in the system and multiple simultaneous migrations can occur. The current limitation of the protocol is the assumption of the use of a global file system.

Measurements of migration costs show that the dominant factor in the migration time is the transfer of the process' virtual address space through the network. This bottleneck has been addressed by system-level process migration implementations. Unfortunately for user-level implementations, unless the OS provides user-level memory management functionality, it would seem that nothing else can done to improve the migration performance.

Though the migration mechanism requires processes to be frozen for some time, the important thing to realize is that this very same mechanism allows PVM applications access to machines they couldn't have used otherwise. It is now possible to have long-running applications execute on a more powerful virtual machine owned by someone else without worrying about getting in the way of the owner. Also, machine owners will likely allow others to use of their machines knowing they will regain dedicated access whenever they want it. Thus, despite the cost of migration, the ability to migrate could lead to large gains in overall performance.

As for future work, a lot of things still have to be done to improve migration transparency: non-reliance on a global file system, support for migrating user-installed signal handlers, use of Unix domain sockets for direct communication between tasks on the same host, etc. Support for migrating applications using X-windows will also be studied. Another aspect being considered is the support for fault-tolerance with the use of checkpointing. More long-term goals for MPVM are integration with a global scheduler that would make it convenient to experiment with various scheduling policies. Integration with existing utilities such as batch schedulers (Condor and DQS [25]), tools (Ptools [26]), profilers and debuggers (Xpvm [27]) etc. is

also be being considered. All this work is targeted for the next generation PVM system.

8 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] 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.

[5] K. Al-Saqabi, S. W. Otto, and J. Walpole. “Gang scheduling in heterogenous distributed systems.” Technical report, Dept. of Computer Science and Engineering, Oregon Graduate Institute of Science & Technology, 1994.

[6] M. Litzkow and M. Solomon. “Supporting checkpointing and process migration outside the Unix kernel.” In *Usenix Winter Conference Proceedings*, pages 283–290, San Francisco, CA, 1992.

[7] “PVM version 3.3.0 release-notes,” June 1994. Comes with the PVM distribution from Netlib.

[8] A. Geist, A. Beguelin, 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 1994.

[9] Y. Artsy and R. Finkel. “Designing a process migration facility – the Charlotte experience.” *Computer*, 22(9):47–56, September 1989.

[10] M. M. Theimer, K. A. Lantz, and D. 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.

[11] A. Barak and A. Litman. “MOS — A multicomputer distributed operating system.” *Software — Practice & Experience*, 15(8):725–737, August 1985.

[12] 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.

[13] F. Douglis and J. 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.

[14] F. Douglis and J. Ousterhout. “Transparent process migration: Design alternatives and the Sprite implementation.” *Software — Practice & Experience*, 21(8):757–785, August 1991.

[15] D. S. Milojicic, W. Zint, A. Dangel, and P. 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.

[16] A. Bricker, M. Litzkow, and M. Livny. “Condor technical summary.” Technical report, University of Wisconsin at Madison, October 1991.

[17] E. R. Zayas. “Attacking the process migration bottleneck.” In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 13–24, Austin, Texas, November 8-11 1987.

[18] 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.

[19] R. Konuru, J. Casas, S. Otto, R. Prouty, and J. Walpole. “A user-level process package for PVM.” In *1994 Scalable High-Performance Computing Conference*, pages 48–55. IEEE Computer Society Press, May 1994.

[20] L. Dikken, F. van der Linden, J. Vesseur, and P. Sloot. *Dynamic PVM – Dynamic Load Balancing on Parallel Systems*, volume II: Networking and Tools of *Lecture Notes in Computer Science*, pages 273–277. Springer-Verlag, Munich, Germany, April 1994.

[21] M. M. Theimer and B. Hayes. “Heterogeneous process migration by recompilation.” Technical Report CSL-92-3, Xerox Palo Alto Research Center - CA, 1992.

[22] A. W. Appel and K. Li. “Virtual memory primitives for user programs.” In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, Santa Clara, CA, April 8-11 1991.

[23] K. Harty and D. R. Cheriton. “Application-controlled physical memory using external page-cache management.” In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 187–197, Boston, Massachusetts, October 12-15 1992.

[24] S. Sechrest and Y. Park. “User-level physical memory management for Mach.” In *Mach Symposium Proceedings*, pages 189–199, Monterey, California, November 20-22 1991.

[25] T. Green and J. Snyder. “DQS, a distributed queuing system.” Technical report, Supercomputer Computations Research Institute, Florida State University, April 1993.

[26] W. Gropp and E. Lusk. “Scalable Unix tools on parallel processors.” In *1994 Scalable High-Performance Computing Conference*, pages 56–62. IEEE Computer Society Press, May 1994.

[27] J. 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.