Adaptive Load Migration Systems for PVM

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

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

March 1994

Presenting Author: Steve W. Otto

email: otto@cse.ogi.edu, Phone: 503-690-1486, FAX: 503-690-1553

Abstract

Adaptive load distribution is necessary for parallel applications to co-exist effectively with other jobs in a network of shared heterogeneous workstations. We present three methods that provide such support for PVM applications. Two of these methods, MPVM and UPVM, adapt to changes in the workstation environment by transparently migrating the virtual processors (VPs) of the parallel application. A VP in MPVM is a Unix process, while UPVM defines light-weight, process-like VPs. The third method, ADM, is a programming methodology for writing programs that perform adaptive load distribution through data movement. These methods are discussed and compared in terms of effectiveness, usability, and performance.

1.0 Introduction

Message-passing systems such as PVM [1-3] allow 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 distributed-memory virtual computer. PVM provides facilities for communication and synchronization between processes and for initiating processes on remote nodes. The main advantage of such a system is that it allows the computing power of widely available, general-purpose workstations networks (worknets) to be harnessed for parallel processing. Because these worknets are idle or partially idle much of the time [4], they are an attractive source of effectively free processing power.

Shared, general-purpose worknets have certain key characteristics that affect execution of parallel applications. First, sharing implies that the load on individual processors and the network varies dynamically. This unpredictable variability can drastically degrade parallel application performance. For example, an entire parallel application can slow because one of its processes is executing on a heavily loaded workstation. In addition to CPU availability, other system resources such as physical memory, virtual memory (swap space), and network bandwidth fluctuate and strongly influence the execution of jobs on the system.

Secondly, despite the high level of sharing, the concept of *ownership* is frequently present. In particular, individual workstations, while available across the network, are often owned by a specific user or group. Owners are often willing to allow outside use when idle, but expect high-quality performance the rest of the time. It is essential that the use of “owned” workstations be *unobtrusive*.

Our proposed solution to these problems consists of enhancing systems such as PVM with the ability to dynamically move portions of the parallel application among the workstations in the network. We describe three methods that provide such support. Although the methods and our prototypes use PVM, the underlying concepts are applicable to other message-passing systems, for example, MPI [12,13].

The first system, Migratable PVM (MPVM), uses Unix processes as its virtual processors (as conventional PVM does) and allows the *transparent migration* of these processes. The processes of a PVM application can be suspended on one workstation and subsequently resumed on another workstation without any help

from the application program. The package is source-code compatible with PVM requiring no more than re-compilation and re-linking of PVM applications. Migration events are initiated and controlled by a global scheduler that is external to the application.

The second system, UPVM, is a virtual processor (VP) package that supports multi-threading and transparent migration for PVM applications [5]. The virtual processors are called User Level Processes (ULPs) and can be thought of as light-weight, Unix-like processes that are independently migratable. UPVM also supports a source-code compatible PVM interface often requiring no modification to the application source.

Finally, Adaptive Data Movement (ADM) [6] is an application-level methodology that provides programmers with an infrastructure for developing adaptive computations based on work re-distribution. Unlike MPVM and UPVM that support work distribution by migrating VPs, ADM provides work distribution through data movement by the application.

The rest of this paper is organized as follows. In section 2.0, we describe the packages in more detail. Section 3.0 presents a qualitative comparison of them followed by basic measurements of application performance in section 4.0. We present related work in section 5.0 and summarize in section 6.0.

2.0 Technical Description

Our packages are implemented in the context of PVM. The PVM system consists of a daemon process (*pvmd*) that runs on each workstation, and a run-time library (*pvmlib*) that contains the PVM interface routines. The *pvmd* is responsible for VP creation and control. VPs in PVM are Unix processes (called *tasks*) linked with the *pvmlib*. Each task has a unique task identifier (*tid*) that defines the end points of task-to-task communication.

All of our systems assume the presence of a network-wide “global” scheduler (GS) that embodies decision-making policies for sensibly scheduling multiple parallel jobs. The GS is also responsible for initiat-

ing a migration by signalling the *pvmd*s. The global schedular is part of the larger, Concurrent Processing Environment (CPE), which is an ongoing research effort [3].

2.1 MPVM

MPVM is an extension of PVM to support transparent migration of process-based VPs. Migrating a process of a parallel application involves capturing the current execution state and transferring this state to another host. A process' state, in this case, not only includes its data, heap, stack and register context, but also its state in relation to the entire parallel application. In a message-passing system, this additional state refers to the messages that are being passed between processes.

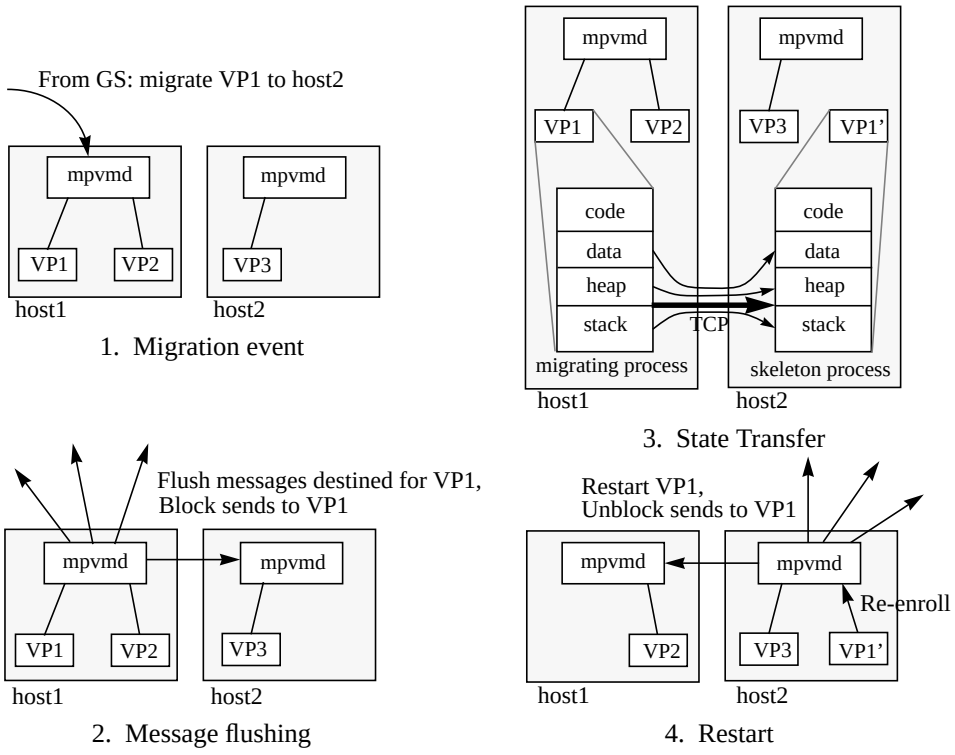

FIGURE 1. MPVM migration. Stages involved in migrating *VP1* from *host1* to *host2*.

The migration protocol used in MPVM can be divided into four major stages: the migration event; message flushing; VP state transfer; and restart (see figure 1).

1. Migration event. The migration of a process is triggered by a migration event. This event could be caused by owner reclamation of a machine or excessively high machine load. Whatever the cause, this event indi-

cates that processes executing on the host where the event occurred must be migrated to other hosts. This migration event causes the GS to send a migrate message to the MPVM daemon (*mpvmd*) on the to-be-vacated-machine (host1 in the figure). The migrate message contains information regarding which process to migrate and where.

2. Message flushing. The *mpvmd,* upon receipt of the migrate message, initiates a message flushing protocol to ensure that: 1) the migrating process has received all messages sent to it prior to migration, and 2) no message is sent to the migrating process for the duration of the migration. This is done by sending a flush message to all other processes informing them of the impending migration. The flush message is acknowledged and from then onwards, a send to the migrating process blocks the sending process.

3. VP state transfer. After flushing the messages, the migrating process is ready to migrate. To accomplish this, a "skeleton" process is started at the destination host. This process has exactly the same code as that of the migrating process (that is, they were executed from the same executable file). A TCP connection is created between the migrating process and the skeleton through which the state of the migrating process is transferred. The skeleton, upon receipt of the state information, assumes it as its own and continues execution accordingly. At this point the skeleton process becomes the migrated process for all practical purposes (VP1' in figure 1).

4. Restart. Before the migrated process can resume in the application execution, it must make itself known to the *mpvmd* on the new host, and send out a restart message to the other processes. This restart message accomplishes two things. First, it unblocks processes blocked on a send to the migrated process. Second, it informs the other processes of its new *tid* making sure that subsequent messages will be sent correctly. Though the process does these things, the application programmer need not concern herself with this protocol. The protocol is done by *mpvmd* and by signal handlers that are transparently linked into the application.

A few more observations are worth noting here. Initiation of the migration is asynchronous with respect to the process to be migrated. That is, a process could be made to migrate at virtually any point of its execution. The only restriction at present is that processes cannot migrate if they are currently executing in the

MPVM run-time library, and the amount of time spent in the library is limited. Secondly, the migration of a process does not necessarily stop the entire parallel application. Only processes sending a message to the migrating process are blocked. Finally, the application executes as if the migration never occurred. This makes MPVM transparent to the application program.

MPVM achieves process migration amongst machines of a like architecture and operating system. Questions concerning heterogeneity, implementation and portability of the package, and transparency to parallel applications are addressed later, in Section 3.0. Some performance figures are given in Section 4.0.

2.2 UPVM

UPVM is a package that supports multi-threading and transparent migration for PVM applications [5]. Though the MPVM package gives a transparent migration capability, UPVM provides a set of “smaller” entities than processes to migrate, allowing load redistribution at a finer granularity.

UPVM defines a new VP abstraction, having some of the characteristics of a thread and some of a process, called a User Level Process (ULP). Like a thread, a ULP defines a register context and a stack. However, ULPs differ from threads in that they also define a private data and heap space (threads share memory with one another). ULPs differ from processes in that their data and heap space is not protected from other ULPs of the same application. That is, ULPs do not define a private protection domain.

From the application programmer's perspective, ULPs look like operating system processes. By convention, ULPs only communicate with each other via message passing. Consequently, existing message-based, parallel applications that use processes can use ULPs with little modification. There are potentially many ULPs per process and they are scheduled by the UPVM library. When a ULP blocks on a message receive, it is de-scheduled and a runnable ULP, if available, is scheduled. Message passing between ULPs on the same process is handled in an efficient manner by the UPVM library, while messages that are destined for ULPs on other processes use a different mechanism.

The fact that each ULP has its own data, stack, and heap significantly simplifies the migration protocol. Since all the state of a ULP is in well-defined locations (unlike threads, whose state could be anywhere in the

encompassing process), it is easy for the UPVM system to find and transfer ULP state. A potential problem with migration concerns pointers in the application program. That is, if a ULP is relocated to a different place in the address space of a process, pointers might have to be modified. To eliminate the need for this, the mapping of a ULP to a set of virtual addresses is made unique across all the processes of the application. For example, consider an application that is decomposed into 5 ULPs across 3 processes, one process per host (see figure 2). If ULP4 is allocated a virtual address region V1 on host3, then V1 is also reserved for ULP4 on all the other hosts, even though it is not present on them.

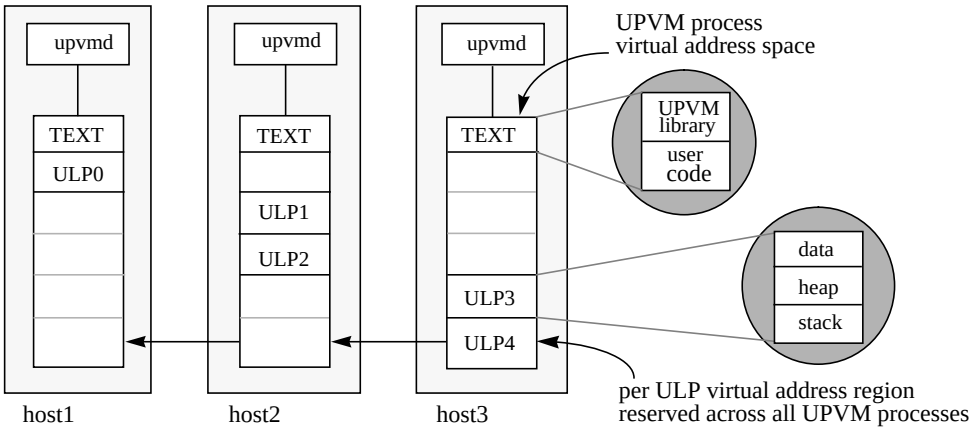

FIGURE 2. ULP System. There is one UPVM process per host per application. ULPs of an application are assigned distinct virtual address regions reserved across all participating processes.

The migration protocol of UPVM goes through four major stages similar to those of MPVM's protocol. These are illustrated in figure 3 and described below.

1. Migration event. The GS sends a migration message directly to the process containing the ULP to be migrated. The process is interrupted, and the register state of the ULP to be migrated is captured.

2. Message flushing. To ensure that no messages are dropped during migration, we send a flush message to all processes and receiving an acknowledgment. The acknowledgment signifies that all messages in transit (for this ULP) have been received. In contrast to MPVM where messages are temporarily blocked, future messages to the ULP are sent directly to the new, target host.

3. VP state transfer. The transfer of ULP state (including unreceived messages) to the target UPVM process is achieved through message sends. Currently, a *pvm_pkbyte() - pvm_send()* sequence is used to transfer this state information though nothing precludes the use of a direct TCP connection as in MPVM. The target UPVM process places the ULP in its allotted virtual address region.

4. Restart. The ULP is placed in the appropriate scheduler queue so that it will eventually execute.

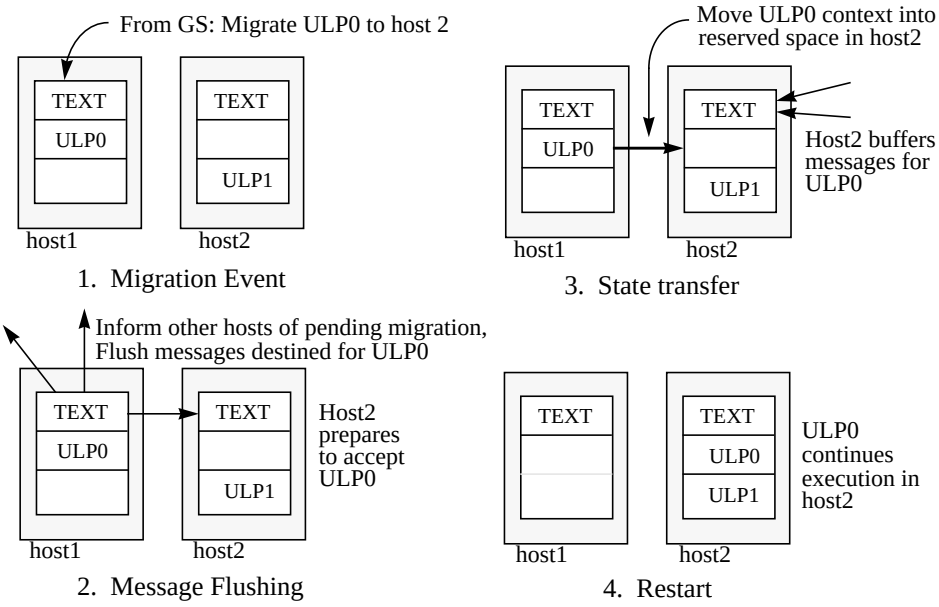

FIGURE 3. UPVM migration. Stages of involved in migrating *ULP0* from *host1* to *host2.*

As in MPVM, UPVM achieves process migration amongst machines of a like architecture and operating system. Questions concerning heterogeneity, implementation and portability of the package, and transparency to parallel applications are addressed later, in Section 3.0. Some performance figures are presented in Section 4.0.

2.3 ADM

In data parallel applications, the amount of work is related to the amount of data to be processed. ADM provides adaptive load distribution for data parallel applications by providing a library of functions designed to respond to migration events, move the data at application level, and then restart the computation.

The ADM approach gives up transparency but has the potential to achieve more accurate load balancing and higher performance.

Three complications arise when extending a data-parallel application to perform adaptive load distribution. First of all, the timing of migration events is inherently *unpredictable* because their source is external to the application. Typically, migration signals come from a global scheduler. The parallel application must be ready to respond appropriately to migration events at arbitrary times.

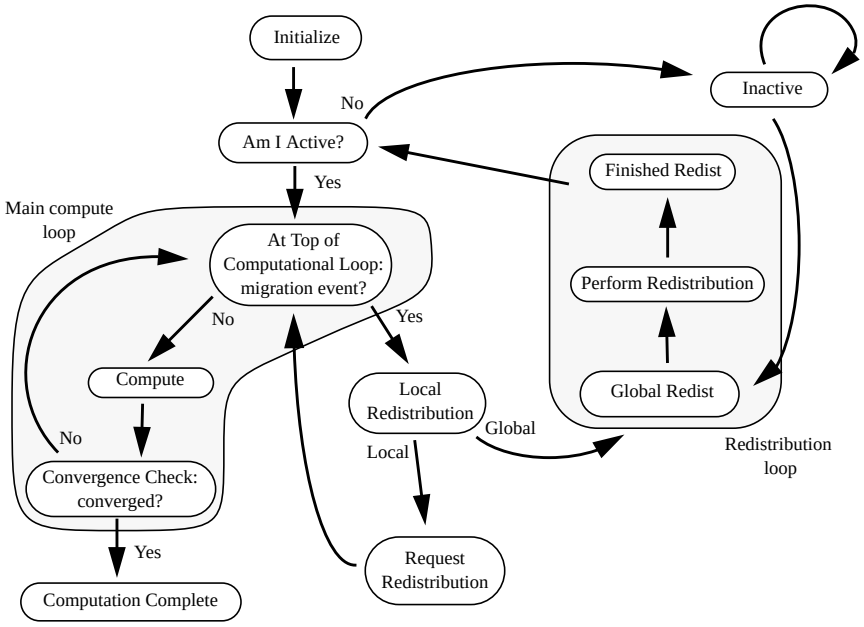

FIGURE 4. The finite-state machine program for ADM Opt.

Secondly, the application must respond *rapidly* to migration events so as to be effective and unobtrusive in capturing idle cycles. Rapid response really means two things: when a migration signal comes, the application should quickly suspend its computation and begin the migration process; and, the migration process itself should be rapid so that the parallel application can quickly resume. This usually implies that migration checks and code to perform redistribution are embedded within the inner computational loops of the program.

Finally, *multiple*, simultaneous migration events must be correctly queued and handled. Since migration events occur at arbitrary times, several of them can happen concurrently and the ADM application must properly queue them. This necessitates careful reasoning about the program so that, no matter what the sequence of events, the migration events are not mis-handled or lost.

These requirements have led us into an event-driven programming style. At a coarse level, the program is written as a finite-state machine. Figure 4 shows this structure for an ADM, data-parallel algorithm. The circles are well-defined states, and arcs are possible transitions that a process can make from one state to another. Each process executes such a finite-state machine and the processes need not be in the same state at any given time, though global-consensus algorithms are executed at some points so as to ensure that all processes have entered a certain state. For example, when an ADM, data-parallel program enters a migration state, the partitioning of the data onto processes is completely re-computed in an attempt to achieve the most accurate load balance possible. This requires global participation by all processes of the parallel application. Different paths through the diagram correspond to normal computing, migration event and load redistribution, and inactivity when a process has no data over which to compute.

Our experience with the ADM approach for data-parallel applications has been that the approach is feasible, but becomes surprisingly complex when the three requirements described above are imposed, a finite-state program such as that in figure 4 results, and great care must be taken to ensure correctness.

3.0 Qualitative Comparison

In supporting adaptive load distribution, the three approaches described represent different trade-offs in application transparency, support for heterogeneity, the granularity of the redistributable work-unit, and the method's portability to new platforms. This section compares current implementations of the three methods with respect to each of these issues.

3.1 Portability

How much effort does it take to port the system to a new platform?

3.1.1 MPVM

MPVM was first implemented for HP-PA workstations running the HP-UX operating system. Subsequently, the system was ported to the SPARC architecture running SUNOS 4.X. The implementation tries to be machine-independent. The migration mechanism, however, is somewhat machine and operating system dependent. We have attempted to limit the dependence on the OS by using generic features found in most versions of Unix [7]. As long as a process can take a snap-shot of its register context and determine the extents of its writable data, heap, and stack space at run-time, porting is not difficult. Most, if not all Unix workstations provide this capability.

3.1.2 UPVM

UPVM runs on HP-PA workstations running the HP-UX operating system. Porting UPVM to a new architecture is a substantial task. For example, the procedure calling conventions of the OS need to be understood. These conventions determine the general and floating point registers that must be saved and restored in a ULP context switch. Also, since ULPs are laid out in distinct regions of a process virtual address space, the virtual memory layout, as defined by the OS, must be understood.

3.1.3 ADM

Since ADM achieves migration through data movement alone, it is, in a sense, truly portable. However, unlike MPVM and UPVM, using ADM on another application requires a re-design of the application. The portability of ADM is therefore application based rather than platform based.

3.2 Transparency

How much does an application programmer need to know about the system in order to use it?

3.2.1 MPVM

Implementing the migration mechanism outside the operating system impacts MPVM's capacity to be completely migration transparent to the application programmer. Because we aren't modifying the underlying

operating system, state information such as process IDs and pending signals *cannot* be preserved on migration. Additional limitations involve: the use of IPC mechanisms (that is, sockets, shared memory, etc.) used outside the MPVM run-time library, timers, process creationl functions such as *fork*() and *exec*(), memory mapped files, and shared libraries. The application programmer must be cautious regarding the use of these facilities with MPVM. A PVM application that does not rely on the above-mentioned limitations should only need re-compilation and re-linking for them to run under MPVM.

3.2.2 UPVM

As with MPVM, migration under UPVM is not completely transparent to the application programmer. Developing applications in UPVM has the same restrictions as that of MPVM in the use of the operating system interface. In addition, there are two more restrictions. One is that only SPMD style applications are currently supported. Secondly, multiple ULPs reside within a single process by dividing the process' virtual address space among *all* the ULPs. This puts a limit on the number of ULPs that could be created depending on the memory requirements (data, stack, and heap) of each ULP. The advent of 64-bit architectures may alleviate this issue.

3.2.3 ADM

As an application level method, an ADM based program cannot help but be programmer dependent. Although MPVM and UPVM also rely on the application programmer for their correctness, the dependence is at different levels. In MPVM and UPVM, its the use of certain system facilities that's the issue. In ADM the issue is at the algorithmic level. That is, the algorithm of an application has to be re-worked for it to use ADM.

3.3 Heterogeneity

How effective is the method in utilizing all the machines in a network?

3.3.1 MPVM

MPVM supports a limited form of heterogeneity in that tasks can be started on hosts of different architectures. However, an MPVM task can only migrate to another migration compatible host. A migration compatible host is one that has similar, if not the same, characteristics of the machine the MPVM task is

migrating from. The problem stems from the fact that non-compatible hosts define process state differently and there is no clear way of translating this state from one non-compatible host to another.

3.3.2 UPVM

UPVM has the same restricted heterogenous support as MPVM. That is, migration can only occur between migration-compatible hosts.

3.3.3 ADM

Heterogeneity is the real strength of ADM. ADM approaches the heterogeneity issue by distributing work as data rather than as processes. Migration takes the form of data redistribution. The main motivation behind ADM is that data can be moved from one machine to another and across platforms with relative ease.

3.4 Redistribution granularity

What is the granularity at which work can be migrated or redistributed?

3.4.1 MPVM

Of the three methods, MPVM is the most coarse-grained in redistributing work. MPVM distributes work at the granularity of whole processes, making it hard to achieve accurate load balance.

3.4.2 UPVM

UPVM is an attempt to resolve the coarse-granularity problem of MPVM. By employing ULPs as the unit of migration, UPVM has the ability to distribute work at a finer granularity. This leads to the ability to achieve better load balance.

3.4.3 ADM

This is another strength of ADM-based applications. Since the load distribution is based on the movement of data, it can potentially achieve ideal load balance. The ADM model itself does not impose any restriction on the precision at which data may be partitioned. The application programmer is free to use whatever precision is most appropriate for the particular application. Thus the application, not the model, limits the accuracy with which the data can be allotted to the heterogeneous processors

4.0 Quantitative Evaluation

Since the main goal of the three systems is to achieve unobtrusive and efficient parallel computation, we use three basic measures in characterizing their performance. These measures are:

1. Inherent method overhead. How much overhead does an application incur when using one of the migration systems as compared to a straightforward implementation (for example, using standard PVM)?
2. Obtrusiveness. What is the time taken to from the instant the migration event was received to the instant the parallel job is "off" the processor?
3. Migration cost. What is the time taken from the instant the migration event was received to the instant when the migrated unit of work is integrated back into the parallel job? This time will be greater than or equal to the obtrusiveness measure.

Since these measures are application dependent, we apply the three methods to "Opt," a neural-network classifying application based on conjugate-gradient optimization [8]. Opt is generally employed as a speech classifier utilizing large (500KB to 400MB) training sets as input. A training set consists of a series of floating point vectors. These vectors, called exemplars, represent digitized speech sound. Each sound belongs to a particular speech category, which is included in the vector as a single scalar value. Opt uses a supervised neural-network algorithm involving back-propagation and conjugate-gradient descent to train the neural-network to predict the corresponding category. The basic algorithm is as follows. An initial neural-net, which is simply a (large) matrix of floating point numbers is established and applied to the exemplars so that a gradient is found. The gradient is also a matrix the same size as the neural-net. That gradient is then used to modify the neural-net before it is reapplied to the data to get a new gradient. This two-step process of applying the neural-net to the exemplars and then modifying the net is repeated until error values pass a threshold or a predetermined number of iterations has been performed.

For the tests that we ran, we used a parallel version of Opt suitably built for each of the three systems. The algorithm in Opt itself remained untouched save for changes required for it to run under these systems.

All experiments were conducted on a quiet system of two HP series 9000/720 workstations connected by a 10Mb/sec Ethernet. Each workstation has a PA-RISC 1.1 processor, 64MB main memory, and is running the HP-UX 9.01 operating system. These experiments will be improved for the final version of the paper.

The parallel Opt, called PVM_opt, has one master VP and 2 slave VPs, one on each machine and data is equally distributed among the slaves. The master VP is responsible for computing a new gradient from partial gradients computed by the slaves, applies this gradient to the neural net, and broadcasts the new neural net to the slaves. The slave VPs, on the other hand, applies the new neural net (from the master) to the exemplars to get a new partial gradient which it passes back to the master and the process repeats again. Note that there is a potential imbalance in that one machine would have the master and 1 slave running on it. This imbalance, however, is offset by the fact that execution of the master and the slave are mutually exclusive in time.

4.1 MPVM

4.1.1 Method overhead

The overhead incurred by an application running normally (no migration) on MPVM, can be attributed to three things. One, whenever the application does a call into the MPVM library, flags have to be set to avoid potential re-entrancy problems when trying to migrate a process while it is executing within the library. Second, there's the overhead associated with *tid* re-mapping. Recall that a migrated process has a new *tid*. This, *tid* re-mapping (from old *tid* to new *tid*) must be done for every message sent and received. This is necessary to ensure that correct delivery of messages to processes. Finally, there's the overhead incurred due to the re-implementation of the *pvm_recv*() call. This re-implementation was necessary to accommodate the case of migrating a process that is blocked in the pvm_recv() routine.

TABLE 1.

PVM	**MPVM**
198 sec	199 sec

Table 1. PVM vs. MPVM, showing the effect of any possible overhead during normal (no migration) execution.

Table 1 shows the results of running PVM_opt on PVM versus MPVM for the 9MB training set. As can be seen, the performance of MPVM is identical to that of PVM. For this particular application, there are

two main factors that mask the overhead of MPVM. First, the size of the messages sent between the master and the slave is large. Thus, the cost of sending the data is dominated by the actual data transfer. Second, note that the overhead of MPVM is only incurred when the application goes into the run-time library. Since this is done quite infrequently in PVM_opt, there is not much additional cost.

4.1.2 Obtrusiveness

Obtrusiveness is measured as the time it takes from when a migrate message is received to the time the work is actually removed from the machine. Table 2 gives a summary of measurements taken while migrating PVM_opt using various data sizes. The raw TCP column in the table show timing results of raw data transfer through the network. This provides us with a lower bound on achievable process migration performance.

TABLE 2.

Data size	Raw TCP	Obtrusiveness	Ratio	Migration time
0.6 MB	0.27 sec	1.17	4.3	1.39
4.2	1.88	2.93	1.56	3.15
5.8	2.51	3.90	1.55	4.10
9.8	4.42	5.92	1.34	6.18
13.5	6.17	8.42	1.36	9.25
20.8	10.00	12.52	1.25	13.10

Table 2: This table shows the obtrusiveness and migration cost for migrating a PVM_opt slave VP for various data sizes. Slaves in the experiments get half of the indicated data size. The Raw TCP column provides us a measure of the lower bound at which we can migrate processes. The fourth column gives the ratio of the obtrusiveness time to the raw TCP time.

The difference between the raw TCP and obtrusiveness times is due to message flushing, starting up the skeleton process on the target host, and setting-up the TCP connection. For larger data sizes, the ratio of obtrusiveness time to the raw TCP transfer time approaches 1.0, showing that we are close to the best possible performance.

4.1.3 Migration cost

Migration cost, as measured in this experiment, measures the time it takes from when a migration event is received to when the migrated process re-participates back into the parallel computation from another host. This time is equivalent to the obtrusiveness time plus the restart time. This restart time includes the time to re-enroll to the *mpvmd* on the new host and the time to send out restart messages to the other VPs in the parallel application. Table 2 shows the migration cost for PVM_opt for various data sizes.

4.2 UPVM

Since the package supports only SPMD applications, an SPMD version of the PVM_opt was created. The SPMD opt program retains the same structure as PVM_opt in that one of the VPs exclusively functions as the master and the rest of the VPs execute as slaves. Thus, when SPMD_opt is executed on the 2 nodes, one node will still have a master VP in addition to a slave VP.

At this time, we present results only for SPMD_opt for 0.6 Mb of data. We are currently extending the UPVM prototype to handle large data and tuning the implementation. The final paper will contain the full results.

4.2.1 Method overhead

As in MVPM, the overhead incurred by an application during normal execution can be attributed to the same three factors. That is, the costs for avoiding potential re-entrancy problems in the library, the mapping of application *tid*s into actual *tid*s for message communication and the implementation of *pvm_recv*() call. In addition, UPVM adds extra information for remote messages that results in marginally slower remote communication than MPVM. However, if an application is divided into more than one VP per node, an application will run faster since UPVM optimizes local communication [5].

TABLE 3.

PVM	**UPVM**
4.92 sec	4.75 sec

Table 3. PVM vs. UPVM, showing the effect of any possible overhead during normal (no migration) execution.

Table 3 shows the results for SPMD_opt. Application performance in UPVM is better because the local communication between the master and slave is optimized and offsets the overhead of the method mentioned above. Specifically, instead of copying the PVM message buffer from the sending ULP into the destination ULP, the UPVM library takes advantage of the shared address space and directly “hands-off” the buffer to the destination ULP. Thus, extra copying is avoided.

4.2.2 Obtrusiveness

The obtrusiveness cost, as measured in this experiment, is the time it takes from when a migration event is received to when all the state of the migrating ULP is off-loaded from the source host. The destination host might not yet have received all the state. Table 4 shows the obtrusiveness cost for the data size of 0.6 Mb.

TABLE 4.

Data size	Obtrusiveness	Migration time
0.6 Mb	1.67 sec	6.88 sec

Table 4. Obtrusiveness and migration costs in UPVM

UPVM is more obtrusive than MPVM because of two factors. First, it uses the *pvm_pkbyte*() routine in transferring the ULP state that results in extra memory copies. Secondly, it collects the message buffers used by the migrating ULP and transfers them in a separate operation using a sequence of *pvm_send*() calls.

4.2.3 Migration cost

The migration cost measured in this experiment is the time taken from the beginning of ULP migration at the source host to when the ULP is finally added to the scheduling queue at the destination (see table 4). Given that the obtrusiveness cost is 1.67 seconds, it is surprising that the migration cost is 6.88 seconds. The difference between the costs is much more than expected. We attribute this to the current implementation of the ULP accepting mechanism in UPVM. We are currently working on optimizing the entire migration mechanism to reduce obtrusiveness and migration costs.

4.3 ADM

Opt is fairly simple to convert into a data-parallel program. The central data structure is essentially one-dimensional, and can be easily partitioned among the processors. Also, any reshuffling of the exemplars that occurs during redistribution affects neither the correctness nor the performance of the algorithm. We take advantage of this in the following way. When ADMopt computes a new partitioning of the data, it does not attempt to preserve an ordering among the exemplars. This allows data that is vacating a process to be fragmented and sent to several other processes.

4.3.1 Method Overhead

In the quiet case, ADMopt suffers some performance degradation compared with PVM_opt. We expect this overhead to come from the use of a switch statement to implement the state diagram in the slaves and the master, and the cost of checking the migration event flag at every iteration of the inner compute loop in the slaves. Since exemplars may be reshuffled during redistribution, the ADMopt slaves are also manipulating an extra data structure. This extra data structure is a simple array of flags used to track which exemplars have been processed and which have not. Thus a slave will not incorrectly reprocess any exemplars they receive from another slave after redistribution. Manipulating this data structure adds a conditional statement and an increment of an array value to the inner loop.

TABLE 5.

PVM_opt	**ADMopt**
188 sec	232 sec

Table 5. Quiet-case overhead, PVM_opt versus ADMopt.

PVM_opt is thus 23% faster than ADMopt. We tried to pinpoint the source of the actual overhead in ADMopt by commenting out such things as the flag checks in the inner loops. However, even with the flag checks removed (effectively disabling migration) ADMopt remained slower than PVM_opt. The overhead may arise from the use of the switch statement or perhaps some code restructuring has unintentionally defeated some compiler optimizations.

4.3.2 Obtrusiveness

The cost of obtrusiveness for ADMopt is defined as the time from the moment when the migrating slave first receives the migration event signal to the time when it receives the message from the master indicating that all slaves have finished redistribution. Table 6 shows the obtrusiveness cost of ADMopt.

4.3.3 Migration Cost

The migration cost is essentially that of the obtrusiveness cost, because ADM does not require a restart stage, as opposed to UPVM and MPVM. Since the withdrawing slave is dividing its data among all other active slaves, it will essentially be the last slave to finish.

TABLE 6.

Data size	Migration time
0.6 MB	1.75 sec
4.2	4.42
5.8	5.46
9.8	9.96
13.5	12.41
20.8	21.69

Table 6. Obtrusiveness (same as migration) cost for ADMopt

5.0 Related Work

There are related systems that address load balancing in the context that we are working (shared worknets).

Condor schedules sequential applications on otherwise idle workstations while remaining unobtrusive to their owners [4,9]. Condor differs from the systems presented since it was originally intended for cycle-stealing by sequential applications (though there is ongoing work to extend Condor to support PVM applications).

It advocates checkpoint-based process migration both for unobtrusiveness and fault tolerance, which has some advantages and some disadvantages compared to the “migrate current state” policy we have chosen for MPVM and UPVM. While the checkpoint approach makes migration less obtrusive, there is a cost of taking periodic checkpoints, and there is a file I/O “idempotency” restriction placed on the application since any part of the computation may be executed more than once.

Data Parallel C (DPC) is a parallel programming environment [10] composed of a compiler and runtime that export a SIMD, shared-address-space model operated upon by a user-specified number of VPs. The number of VPs is usually much larger than the number of processors. The multi-computer DPC compiler

translates the SIMD DPC program source into SPMD C code. The SPMD C code is then compiled into an executable image using a C compiler and an OS process is created on each allocated processor. Multiple VPs are then emulated within each process. DPC is closest to our UPVM system. They both separate application-level parallelism from processor availability and make the efficient choice of one process per allocated processor. Also, both systems perform load distribution at the granularity of VPs. They differ however in that 1) DPC provides heterogeneous VP migration, 2) DPC is language based (that is, parallelism is expressed using constructs in the Data Parallel C language), and 3) VP migration can only occur at specific points in the program. Specifically, VP migration is possible only at the beginning or end of code segments that emulate a single VP.

Piranha [11] is a system for adaptive parallel programming implemented on top of Linda. Piranha is very similar to ADM in that the application programmer structures the program in a certain way and writes certain functions that allow the application to adapt to a dynamic environment. While Piranha communicates with other processes through a global memory object (tuple space) which exports a distributed-shared memory model, ADM is directed at applications that communicate via explicit message-passing.

6.0 Summary

This paper has discussed three approaches to achieving adaptive load migration on shared networks of workstations. For each of the approaches, we have built prototype systems that function as intended and are capable of running realistic, scientific applications written for the PVM message-passing interface. The MPVM prototype adds transparent process migration to PVM. When coupled to a global scheduler, it provides a realistic way of achieving adaptive load migration. UPVM can be considered a logical step after MPVM. Since the notion of “processes” are too heavy-weight and are inappropriate as the fundamental, virtual processors of a parallel application, UPVM takes the MPVM approach of transparent migration, but does so with smaller units that we call “user-level processes.” Such “ULPs” have some of the characteristics of threads and some of the characteristics of processes and are transparently migratable. UPVM has the potential of being more effective in load migration since the ULPs allow load redistribution at a much finer granularity than processes. Both MPVM and UPVM are source-code compatible with PVM; applications (usually) need only to be

re-compiled and re-linked to use the systems. Neither system introduces large, extra overheads to the parallel application when it is running in normal, non-migrating mode. Our times for migration (a few seconds for a 1MB job running on an Ethernet-based LAN) are sufficiently encouraging to make the approach viable.

As a point of contrast, we have also investigated the use of application-level approaches to adaptive load redistribution for data-parallel algorithms. The complexity arising from the need to handle unpredictable, multiple migration events in an efficient way is surprising. We have a working application that uses the ADM method and we suggest an event-driven programming style (at a coarse level) to cope with migration complexities.

7.0 References

[1] Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, and Vaidy Sunderam, "Heterogeneous Network Computing," *Proceedings of the Sixth SIAM Conference on Parallel Processing*, 1993.

[2] Jack Dongarra, Al Geist, Robert Manchek, and Vaidy Sunderam, "Integrated PVM Framework Supports Heterogeneous Network Computing," *Computers in Physics*, April 1993, pp. 166-75.

[3] Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, Steve W. Otto, and Jonathan Walpole, "PVM: Experiences, Current Status and Future Directions", in proceedings, *Supercomputing '93*, IEEE Computer Society Press, Los Alamitos, CA, pages 765-6, November 1993.

[4] Michael K. Litzkow, Miron Livny, and Matt W. Mutka, "Condor -- A Hunter of Idle Workstations," *Proceedings of the 8th IEEE International Conference on Distributed Computing Systems*, pp. 104-11, San Jose, CA, June 1988.

[5] Ravi Konuru, Jeremy Casas, Steve W. Otto, Robert Prouty, and Jonathan Walpole, "A User-Level Process Package for PVM," to appear in proceedings, *1994 Scalable High-Performance Computing Conference*, Knoxville, TN, May 1994.

[6] Robert Prouty, Steve W. Otto, and Jonathan Walpole, "Adaptive Execution of Data Parallel Computations on Networks of Heterogeneous Workstations," Technical Report CSE-94-012, Dept. of Computer Science, Oregon Graduate Institute of Science & Technology, 1994.

[7] Michael Litzkow and Marvin Solomon, “Supporting Checkpointing and Process Migration Outside the Unix Kernel,” *Usenix Winter 1992 Technical Conference*, pp. 283-90, January 1992.

[8] Etienne Barnard and Ronald Cole, “A Neural-net Training Program Based on Conjugate Gradient Optimization,” Technical Report CSE-89-014, Oregon Graduate Institute of Science & Technology, 1989.

[9] Allan Bricker, Michael Litzkow, and Miron Livny, “Condor Technical Summary”, Computer Sciences Department, University of Wisconsin - Madison, October 91.

[10] Nenad Nedeljkovic and Michael J. Quinn, “Data-Parallel Programming on a Network of Heterogeneous Workstations”, *Concurrency: Practice and Experience*, volume 5(4), pages 257-268, June 1993.

[11] N. Carriero, D. Galernter, D. Kaminsky, and J. Westbrook, “Adaptive Parallelism with Piranha,” available via anonymous ftp from dept-gw.cs.yale.edu as pub/piranha.ps.Z .

[12] The Message-Passing Interface Forum, “Document for a Standard Message-Passing Interface,” University of Tennessee Computer Science technical report, CS-93-214, November, 1993.

[13] The Message-Passing Interface Forum, “MPI: A Message-Passing Interface,” in proceedings, *Supercomputing ‘93,* IEEE Computer Society Press, Los Alamitos, CA, pp. 878-83.