Processor Virtualization and Migration for PVM

Steve W. Otto*

Abstract

This paper describes research underway to define and develop the next generation of PVM (Parallel Virtual Machine). Future versions of PVM will be modular and open so as to allow interoperability with other packages, such as distributed scheduling systems.

We concentrate on one aspect of the work: providing virtualization of processors and transparent migration mechanisms within the message-passing programming model. Work migration is a key ingredient to allow good scheduling on a large, busy system. Two migration systems will be described. The first is Migratable PVM (MPVM), which allows transparent migration at process granularity amongst homogeneous groups of processors. The system is functional and has run realistic applications. The second system is a multi-threaded version of PVM, where threads are disjoint and do not share data spaces. This again allows transparent migration. Local communication speeds and context-switch times are improved over process-level MPVM. Performance figures and semantic restrictions of both packages are given.

1 Introduction

Concurrent processing applications are evolving into large, complex systems, with many interacting sub-tasks whose resource requirements often differ widely. At the same time, computing environments are becoming more versatile, typically consisting of multiprocessors, vector supercomputers, and graphics engines, in addition to general purpose workstations. Effectively harnessing this heterogeneous collection of resources, and enabling the most effective use of specialized capabilities, requires a supporting (systems) software architecture that presents a virtual Concurrent Processing Environment (CPE) to the user.

The PVM (Parallel Virtual Machine) system virtualizes a heterogeneous collection of computers into a distributed-memory, message-passing, parallel computer [2, 6]. PVM allows concurrent processing on heterogeneous systems interconnected by fast networks and has demonstrated the technical and economic viability of this computing model. Its simplicity and robustness have encouraged the development of a large number of PVM applications, and the list of PVM users is long and growing rapidly. PVM spans network computers, cluster computers, and tightly-coupled massively-parallel processors (MPPs). Groups at the University of Tennessee, Oak Ridge National Laboratory, Carnegie Mellon University, and Oregon Graduate Institute have begun research to enhance the functionality of PVM, with emphasis on: migration capability and interfaces to distributed schedulers and resource managers; tools for program development and performance monitoring; and a new interface adding multimedia and visualization capabilities to PVM [1]. Some of this functionality will be extended to include MPI, the standardized message-passing interface [7].

*Dept of Computer Science and Engineering, Oregon Graduate Institute of Science & Technology, 20000 NW Walker Road, P. O. Box 91000, Portland OR 97291-1000, otto@cse.ogi.edu

1.1 An Open Framework

The new version of PVM will form the "kernel" of the CPE. Rather than build all the above capabilities directly into the kernel, we take a modular approach. Once interfaces are defined between the modules of the CPE, the prototype environment can coexist with related software systems from other research groups or vendors. The distributed scheduler, for example, will be a separate module with a well-defined interface to the CPE kernel. We plan to provide a simple, default scheduler, but one could use others. This "open-systems" approach addresses heterogeneity at the system software level as well as at the operating system and hardware levels. Figure 1 illustrates the overall architectural framework by identifying the generic modules that we expect to exist in a CPE, the interfaces among them, and the interfaces to other system components such as the operating system and file system.

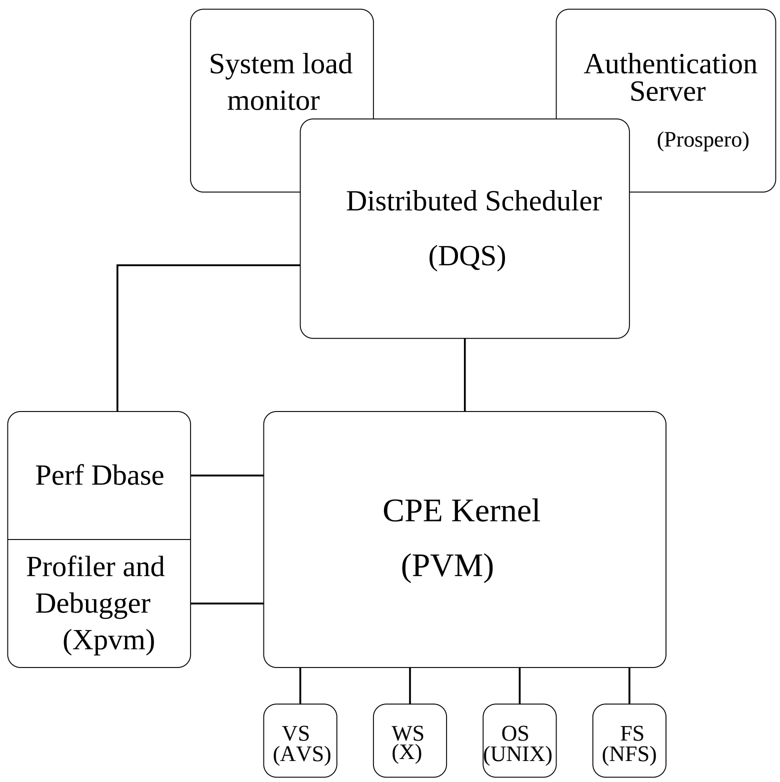

FIG. 1. *The software architecture of the CPE. This diagram shows the main modules of a CPE as large boxes and the defined interfaces between them as lines. The smaller boxes such as VS (visualization system) are existing modules that we cannot modify. Note that this picture illustrates the software architecture and not processes – several of the modules here are distributed and correspond to many executing processes.*

1.2 The CPE as Middleware

The CPE kernel is "middleware" [3], providing a well-defined, virtual environment through which concurrent applications access system resources. Besides message passing, the CPE kernel also provides its own process control abstractions, internal interfaces to other CPE modules, and external interfaces to the operating system, file system, windowing system and visualization environment. An important goal of the CPE kernel is to provide a consistent single-system image to concurrent applications in a heterogeneous distributed execution environment.

PVM currently defines a simple interface for message-passing and process control, but lacks some of the functionality desired for the CPE kernel, and leaves certain interfaces undefined. For example, PVM currently encourages applications to directly access operating system and file system facilities, and consequently does not provide a single system image to applications (e.g., file names are location dependent). Similarly, the CPE kernel will export an explicit scheduling interface such that resource allocation can be done globally rather than on a per application basis.

Due to the severe constraints of portability across operating systems, creating efficient and functional middleware such as the CPE is a challenge. See [4] for some thoughts on what operating systems should provide to support such systems.

1.3 Distributed Scheduler and Authentication

Current PVM includes many process creation and management tasks that are more properly left to a separate scheduling system. The CPE kernel will give up some of this functionality to the scheduler. Users may add to or delete from the pool of hosts on which a concurrent application is to execute only by negotiating with the scheduler. Since the scheduler has the necessary global information required to make sensible resource allocation and job management decisions, migration decisions are also made by it.

Distributed schedulers in a large-scale heterogeneous environments require additional functionality to access resources in multiple administrative domains. Rather than build authentication and accounting functionality directly into the scheduler, however, we will use existing systems such as the Prospero resource manager [11, 12].

2 Migration

Processor virtualization is an attractive goal because it frees application programmers from the burden of managing physical processor location and availability. Virtual processors (VPs) allow programmers to think solely in terms of the parallelism within their application. Processor virtualization also improves system resource utilization because it allows systems software to transparently adapt to changes in processor availability, preemption, and load imbalance. Support for dynamic reallocation is useful in large multicomputers and essential in shared workstation environments. The remainder of this paper describes two processor virtualization and migration systems that we have built.

The first system, Migratable PVM (MPVM), uses Unix processes as its virtual processors (as does conventional PVM) and allows the transparent migration of these processes [5]. 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 package that supports multi-threading and transparent migration for PVM applications [8]. 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.

FIG. 2. *MPVM migration. Illustrated are the 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 2).

1. Migration event. The migration of a process is triggered by a migration event. This event indicates 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 the figure).

4. Restart. Before the migrated process can rejoin 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 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 [9, 10]. 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.

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 creation 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 it to run under 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 straightforward way of translating this state from one non-compatible host to another.

3.1 MPVM Performance

The overhead incurred by an application running normally (no migration) with 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 an overhead associated with tid re-mapping. Finally, there is an overhead incurred due to a re-implementation of the pvm_recv() call. This was necessary to accommodate the case of migrating a process that is blocked in the pvm_recv() routine. These factors cause a minor increase in message-passing times for MPVM versus PVM.

Obtrusiveness is defined as the time it takes from when a migrate message is received to the time the work is actually removed from the machine. Table 1 gives a summary of measurements taken while migrating PVM_opt[1] using various data sizes. The raw TCP column in the table shows timing results of raw data transfer through the network. This provides us with a lower bound on achievable process migration performance.

Table 1

This table shows the obtrusiveness and migration cost for migrating a PVM_opt slave VP for various data sizes. 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.

Data size	Raw TCP	Obtrusiveness	Ratio	Migration Time
0.6 MB	0.27 sec	1.17 sec	4.3	1.39 sec
4.2	1.88	2.93	1.56	3.15
9.8	4.42	5.92	1.34	6.18
20.8	10.00	12.52	1.25	13.10

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 large data, the ratio of obtrusiveness time to the raw TCP transfer time approaches 1.0.

Migration cost measures the time it takes from when a migration event is received to when the migrated process rejoins the parallel computation from another host. This time is equivalent to the obtrusiveness time plus the restart time.

4 User-level Process PVM (UPVM)

UPVM is a package that supports multi-threading and transparent migration for PVM applications [8]. 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. Context switching and on-processor (local) communication times are also greatly improved over MPVM.

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.

[1] PVM_opt is a parallel, neural-net training application.

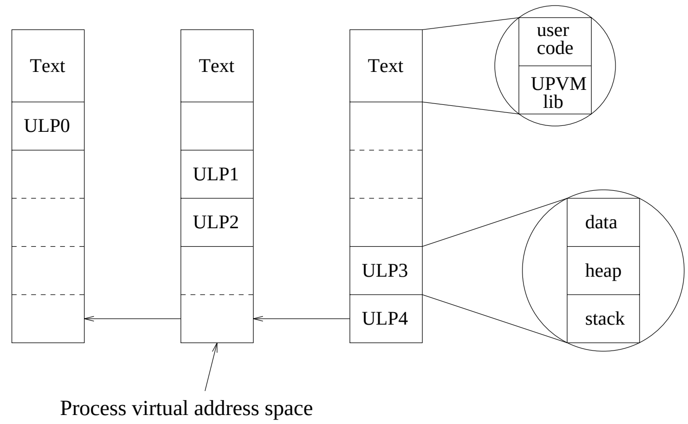

FIG. 3. *Address space layout of the UPVM system. There is one UPVM process per host per application. ULPs of an application are assigned distinct virtual address regions, reserved across all participating processes.*

From the application programmer's perspective, ULPs look much 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 descheduled 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), it is easy for the 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 3). 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.

The migration protocol of UPVM goes through four major stages similar to those for MPVM. These are 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 ULP state (including unreceived messages) is transferred to the target UPVM process. This is done using conventional PVM. The target UPVM

process places the ULP in its allotted virtual address region. Message queues are also modified so that pairwise in-order message delivery is preserved.

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

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.

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 — SPMD leads to many simplifications for UPVM. 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.

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

To evaluate the performance of the UPVM package, we give the results of micro-benchmarks for context switch, local communication, and remote communication. The goal is to measure the costs of the primitive operations provided by UPVM. All experiments were conducted on two HP series 9000/720 workstations that were otherwise idle, connected over a 10Mb/sec Ethernet. Each of the workstations has a PA-RISC 1.1 processor, 64 MB main memory, and is running the HP-UX 9.01 operating system.

The context switch benchmark measures the time taken for one VP (an OS process or ULP) to yield to another of the same kind. For comparison purposes, the cost of executing a null procedure call on the HP-UX workstation is 0.65 micro-seconds. Table 2 gives the context switch cost of ULPs and OS processes, both in absolute time and as a ratio to null procedure call cost.

TABLE 2

Context switch costs (absolute and relative to null procedure call time) for UPVM.

Type	Cost (micro-seconds)	Ratio
ULP switch	4.74	7.30
UNIX switch	195.00	300.46

Isolating the process context switch cost in a portable manner is extremely difficult, since there is no equivalent of a yield-to-another-process system call on UNIX. Our solution to this problem was to use Ousterhout's context switch benchmark [13]. In this case, we calculate half the time taken by two UNIX processes to alternately read and write one byte from a pair of pipes. This implies that the UNIX process switch cost given in Table 2 includes the cost of reading and writing one byte from a pipe in addition to the true process switch costs. However, even if we consider only half of the observed process switch costs, the ULP switch is still more than an order of magnitude faster.

The local communication benchmark measures the round-trip message communication cost between two VPs. The benchmark is compiled with the PVM library and then with UPVM, yielding two different executables. In the case of PVM, the local communication cost measured is between two UNIX processes on the same node. In the case of UPVM, the cost measured is between two ULPs that are executing within the same UNIX process. The numbers in Table 3 are half the round-trip cost. We assume that this closely approximates the one-way communication cost. The local communication cost of UPVM is around an order of magnitude better than that of PVM.

TABLE 3

Local communication costs for UPVM.

Message size(bytes)	PVM(ms)	UPVM(ms)
0	1.40	0.12
1000	1.85	0.14
10000	6.55	0.39
100000	47.36	5.55

Since UPVM uses PVM for remote communication, we expected a marginal increase in the cost of the remote communication. This increase is visible in Table 4, which shows that UPVM costs are about 3.5 %, 3% and 1% higher for 1K, 10K and 100K message sizes respectively.

TABLE 4

Remote communication costs for UPVM.

Message size(bytes)	PVM(ms)	UPVM(ms)
0	2.65	2.80
1000	4.01	4.15
10000	17.06	17.60
100000	144.70	146.36

Finally, Table 5 shows the costs for migration of ULPs. As was the case for MPVM, the figures are fairly close (60–70%) to the raw TCP speeds available on the 10 MB/sec ethernet link used for the experiment.[2]

TABLE 5

Migration time as a function of data size for UPVM.

Data Size (MB)	Time (sec)
.25	0.44
.5	0.80
1.0	1.57
2.0	3.09

Acknowledgements

Jeremy Casas wrote the MPVM system and Ravi Konuru implemented UPVM. I thank them and the other members of our research group, Jonathan Walpole, Robert Prouty,

[2] Note that "data size" is defined differently here than for the MPVM experiment — there, it meant the total size of the data set, here it means the per ULP size.

Khaled Al Saqabi, Dan Clark, and Jon Inouye, for helpful discussions. Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, and Jonathan Walpole made key contributions to the overall design of the Concurrent Processing Environment.

References

[1] 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, 1993, pp. 765–6.

[2] A. L. Beguelin, J. J. Dongarra, A. Geist, R. J. Manchek, and V. S. Sunderam, *Heterogeneous network computing*, in Sixth SIAM Conference on Parallel Processing, 1993.

[3] P. Bernstein, *Middleware: An architecture for distributed system services*, Tech. Rep. CRL93/6, Cambridge Research Lab, Digital Equipment Corp., March 1993. Submitted for publication.

[4] A. P. Black and J. Walpole, *Objects to the rescue!*, tech. rep., Oregon Graduate Institute of Science & Technology, May 1994. Position paper for SIGOPS 1994 European Workshop.

[5] J. Casas, R. Konuru, S. W. Otto, R. Prouty, and J. Walpole, *Adaptive load migration systems for PVM*, tech. rep., March 1994. Submitted to Supercomputing '94.

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

[7] M. P. I. Forum, *MPI: A message-passing interface standard*, computer Science Dept. Technical Report CS-94-230, University of Tennessee, Knoxville, TN, April 1994. (To appear in the International Journal of Supercomputer Applications, Volume 8, Number 3/4, 1994).

[8] R. Konuru, J. Casas, S. W. Otto, R. Prouty, and J. Walpole, *A user-level process package for PVM*, in 1994 Scalable High-Performance Computing Conference, IEEE, May 1994. To appear.

[9] M. Litzkow, M. Livny, and M. Mutka, *Condor — a hunter of idle workstations*, in Proceedings of the 8th International Conference on Distributed Computing Systems, San Jose, CA, June 1988, IEEE, pp. 104–111.

[10] M. Litzkow and M. Solomon, *Supporting checkpoint and process migration outside the unix kernal*, in Usenix Winter Conference, 1992.

[11] B. C. Neuman, *Prospero: A tool for organizing Internet resources*, Electronic Networking: Research, Applications and Policy, 2 (1992).

[12] B. C. Neuman and S. Rao, *Resource management for distributed parallel systems*, in Proceedings of the 2nd Internationational Symposium on High Performance Distributed Computing, July 1993.

[13] J. K. Ousterhout, *Why aren't operating systems getting faster as fast as hardware?*, in Proceedings of the Summer 1990 USENIX Conference, Anaheim, CA, June 1990, pp. 247–256.