# PVM and MPI: a Comparison of Features

G. A. Geist
J. A. Kohl
P. M. Papadopoulos *

May 30, 1996

**Abstract**

This paper compares PVM and MPI features, pointing out the situations where one may be favored over the other. Application developers can determine where their application most likely will run and if it requires particular features supplied by only one or the other of the APIs.

MPI is expected to be faster within a large multiprocessor. It has many more point-to-point and collective communication options than PVM. This can be important if an algorithm is dependent on the existence of a special communication option. MPI also has the ability to specify a logical communication topology.

PVM is better when applications will be run over heterogeneous networks. It has good interoperability between different hosts. PVM allows the development of fault tolerant applications that can survive host or task failures. Because the PVM model is built around the *virtual machine* concept (not present in the MPI model), it provides a powerful set of dynamic resource manager and process control functions.

Each API has its unique strengths and this will remain so into the foreseeable future. One area of future research is to study the feasibility of creating a programming environment that allows access to the virtual machine features of PVM and the message passing features of MPI.

## 1. Introduction

The recent emergence of the MPI (Message Passing Interface) specification [2] has caused many programmers to wonder whether they should write their applications in MPI or use PVM (Parallel Virtual Machine) [3]. PVM is the existing de facto standard for distributed computing and MPI is being touted as the future message passing standard. A related concern of users is whether they

*This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract DE-AC05-96OR22464 with Lockheed Martin Energy Research Corporation

should invest the time and effort to rewrite their existing PVM applications in MPI.

In this paper we address these questions by comparing the features supplied by PVM and the features supplied by MPI and showing under which situations one API might be favored over another. Programmers can then assess the needs of their application and decide accordingly.

Computer vendors are driven by the needs of their buyers, some of whom insist on PVM and others on MPI. Therefore, all the major vendors have now committed resources to provide both PVM and MPI on their systems. This removes concerns of availability and support for either PVM or MPI and allows us to concentrate on the features and capabilities that distinguish them.

We have been involved in the design of both PVM and MPI. The design process was quite different in both cases as were the focus and goals of the designs. Some background material will help better illustrate how PVM and MPI differ and why each has features the other does not.

## 2. Background

The development of PVM started in the summer of 1989 when Vaidy Sunderam, a professor at Emory University, visited Oak Ridge National Laboratory to do research with Al Geist on heterogeneous distributed computing. They needed a framework to explore this new area and so developed the concept of a Parallel Virtual Machine (PVM) [8, 4]. In 1991, Bob Manchek (a research associate at the University of Tennessee) joined the research and implemented a portable, robust version of the PVM design (PVM 2.0). Jack Dongarra, who was also involved in our heterogeneous distributed computing research, was instrumental in making PVM 2.0 publically available. The use of PVM grew rapidly worldwide as scientists spread the word of the utility of this software to do computational research.

Central to the design of PVM was the notion of a "virtual machine" – a set of heterogeneous hosts connected by a network that appears logically to the user as a single large parallel computer. One aspect of the virtual machine was how parallel tasks exchanged data. In PVM this was accomplished using simple message-passing constructs. There was a strong desire to keep the PVM interface simple to use and understand. Portability was considered much more important than performance for two reasons: communication across the internet was slow; and, the research was focused on problems with scaling, fault tolerance, and heterogeneity of the virtual machine.

As the PVM user base grew into the thousands, a conscious effort was made to keep the PVM API backwards compatible so that all existing PVM applications would continue to run unchanged with newer PVM versions. All PVM version 2 releases are backwards compatible, as are all PVM version 3 releases. PVM 3.0 was released in 1993 with a completely new API. The API change and new design were required to enable a PVM application to run across a virtual machine composed of multiple large multiprocessors.

The PVM API has continuously evolved over the years to satisfy user requests for additional features and to keep up with the fast-changing network and computing technology. One example of a user-requested feature was the addition of interfaces that allow third-party debuggers and resource managers to be seamlessly incorporated into the virtual machine [9, 6]. Examples of technology driven changes include the ability for PVM to transparently utilize shared memory and high-speed networks like ATM to move data between clusters of shared-memory multiprocessors.

In contrast to the PVM API, which sprang from and continues to evolve inside a research project, the MPI-1 API was specified by a committee of about 40 high performance computing experts from research and industry in a series of meetings in 1993-1994. The impetus for developing MPI was that each Massively Parallel Processor (MPP) vendor was creating their own proprietary message-passing API. In this scenario it was not possible to write a portable parallel application. MPI is intended to be a standard message-passing specification that each MPP vendor would implement on their system. The MPP vendors need to be able to deliver high performance and this became the focus of the MPI design. Given this design focus, MPI is expected to always be faster than PVM on MPP hosts. Even so, two recent comparison studies show that PVM and MPI have very comparable performance on the Cray T3D and IBM SP-2 [10, 5].

MPI-1 contains the following main features:

- A large set of point-to-point communication routines (by far the richest set of any library to date),
- A large set of collective communication routines for communication among groups of processes,
- A communication context that provides support for the design of safe parallel software libraries,
- The ability to specify communication topologies,
- The ability to create derived datatypes that describe messages of noncontiguous data.

MPI-1 users soon discovered that their applications were not portable across a network of workstations because there was no standard method to start MPI tasks on separate hosts. Different MPI implementations used different methods. In 1995 the MPI committee began meeting to design the MPI-2 specification to correct this problem and to add additional communication functions to MPI including:

- MPI_SPAWN functions to start both MPI and non-MPI processes,
- One-sided communication functions such as *put* and *get*,

- Nonblocking collective communication functions
- Language bindings for C++.

The MPI-2 draft specification is scheduled to be finished by January 1997. In its present form the MPI-2 draft adds an additional 120 functions to the 128 functions specified in the MPI-1 API. This makes MPI a much richer source of communication methods than PVM.

Will there be an MPI-3? It is too soon to tell but there are some useful features available in PVM that will not be available in MPI-2. MPI users may request an MPI-3 specification that allows them to create fault tolerant applications, interoperates among different MPI implementations, and dynamically determines available resources.

In the sections that follow, we discuss the major feature differences between MPI and PVM.

## 3. Portability versus Interoperability

Heterogeneity is becoming increasingly important for high performance computing. Massively parallel processors appear to be a dying breed, leading scientists with serious computational needs to look towards clusters of smaller multiprocessors connected by new high-speed networks. Many organizations already use a variety of different computing systems in the form of different personal computers or workstations on their employees' desks. Integrating these desktop machines and utilizing their unused cycles can be an effective way of obtaining reasonable computational power. Parallel software systems therefore need to accommodate execution on many different vendor platforms. In addition to running on MPPs, PVM and some implementations of MPI already work on networked clusters of machines.

The MPI interface was developed with the intent of encompassing all of the message-passing constructs and features of various MPP and networked clusters so that programs would execute on each type of system. The *portability* achieved by MPI means that a program written for one architecture can be copied to a second architecture, compiled and executed without modification.

PVM also supports this level of portability, but expands the definition of portable to include *interoperable*. PVM programs similarly can be copied to different architectures, compiled and executed without modification. However, the resulting PVM executables can also *communicate with each other*. In other words, an MPI application can run, as a whole, on any single architecture and is portable in that sense. But a PVM program can be ported heterogeneously to run cooperatively across any set of different architectures at the same time (i.e. interoperate). While the MPI standard does not prohibit such heterogeneous cooperation, it does not require it. Nothing in the MPI standard describes cooperation across heterogeneous networks and architectures. And there is no impetus for one vendor to make its MPI implementation slower in order to allow a user to use another vendor's machine. None of the existing MPI implementations can interoperate.

There would need to be another standard – one for interoperability. MPI would need to check the destination of every message and determine if the destination task is on the same host or on some other host. If it is on some other host, with that vendor's MPI implementation, the message must be converted into a format that can be understood by the other MPI version.

The lack of MPI's flexibility in this scenario comes from the pre-emptive priority of performance in its design. The best way to make a message-passing program fast on any single architecture is to streamline the underlying library to use native hardware and eliminate any unnecessary waste for that architecture. For example, if a certain architecture provides some built-in mechanism for broadcasting messages, the MPI implementation should use that mechanism directly. However, using that native mechanism makes it more difficult and less efficient to use broadcast functions to send messages to hosts of different vendors that do not support the mechanism.

The PVM solution to this problem is to sacrifice some performance in favor of the flexibility to communicate across architectural boundaries. When communicating locally or to another host of identical architecture, PVM uses the native communication functions just like MPI. When communicating to a different architecture, PVM uses the standard network communication functions. Because the PVM library must determine from the destination of each message whether to use the native or network communication, there is some small overhead incurred.

PVM and MPI also differ in their approach to language interoperability. In PVM, a C program can send a message that is received by a Fortran program and vice-versa. In contrast, a program written in C is not required by the MPI standard to communicate with a program written in Fortran, even if executing on the same architecture. This restriction occurs because C and Fortran support fundamentally different language interfaces, causing difficulty in defining a consistent standard interface that covers both. The MPI decision was to not force the two languages to interoperate.

## 4. Virtual Machine

PVM is built around the concept of a *virtual machine* which is a dynamic collection of (potentially heterogeneous) computational resources managed as a single parallel computer. The virtual machine concept is fundamental to the PVM perspective and provides the basis for heterogeneity, portability, and encapsulation of functions that constitute PVM.

It is the virtual machine concept that has revolutionized heterogeneous distributed computing by linking together different workstations, personal computers and massively parallel computers to form a single integrated computational engine. In contrast, MPI has focused on message-passing and explicitly states that resource management and the concept of a virtual machine are outside the scope of the MPI (1 and 2) standard.

### 4.1. ***Process Control***

Process control refers to the ability to start and stop tasks, to find out which tasks are running, and possibly where they are running. PVM contains all of these capabilities. In contrast MPI-1 has no defined method to start a parallel application. MPI-2 will contain functions to start a group of tasks and to send a kill signal to a group of tasks (and possibly other signals as well).

Some basic resource query capability is important in order to know how many tasks can be started on the available (possibly dynamic) computing resources. In this regard, PVM has a rich set of resource control functions.

### 4.2. ***Resource Control***

In terms of resource management, PVM is inherently dynamic in nature. Computing resources, or "hosts," can be added or deleted at will, either from a system "console" or even from within the user's application. Allowing applications to interact with and manipulate their computing environment provides a powerful paradigm for load balancing, task migration, and fault tolerance. The virtual machine provides a framework for determining which tasks are running and supports naming services so that independently spawned tasks can find each other and cooperate.

Another aspect of virtual machine dynamics relates to efficiency. User applications can exhibit potentially changing computational needs over the course of their execution. Hence, a message-passing infrastructure should provide flexible control over the amount of computational power being utilized. For example, consider a typical application which begins and ends with primarily serial computations, but contains several phases of heavy parallel computation. A large MPP need not be wasted as part of the virtual machine for the serial portions, and can be added just for those portions when it is of most value. Likewise, consider a long-running application in which the user occasionally wishes to attach a graphical front-end to view the computation's progress. Without virtual machine dynamics, the graphical workstation would have to be allocated during the entire computation. MPI lacks such dynamics and is, in fact, specifically designed to be static in nature to improve performance. There is clearly a trade-off in flexibility and efficiency for this extra margin of performance.

Aside from more tangible effects, the virtual machine in PVM also serves to encapsulate and organize resources for parallel programs. Rather than leaving the parallel programmer to manually select each individual host where tasks are to execute and then log into each machine in turn to actually spawn the tasks and monitor their execution, the virtual machine provides a simple abstraction to encompass the disparate machines. Further, this resource abstraction is carefully layered to allow varying degrees of control. The user might create an arbitrary collection of machines and then treat them as uniform computational nodes, regardless of their architectural differences. Or the user could traverse the increasing levels of detail and request that certain tasks execute on machines with particular data formats, architectures, or even on an explicitly named machine.

The MPI standard does not support any abstraction for computing resources and leaves each MPI implementation or user to customize their own management scheme. Though such a customized scheme can ultimately be more convenient for a particular user's needs, the overhead to construct the scheme counters the gains. With PVM, this customization is always possible using the existing "virtual machinery," should the user desire more control.

### 4.3. ***Topology***

Although MPI does not have a concept of a virtual machine, MPI does provide a higher level of abstraction on top of the computing resources in terms of the message-passing topology. In MPI a group of tasks can be arranged in a specific logical interconnection topology. Communication among tasks then takes place within that topology with the hope that the underlying physical network topology will correspond and expedite the message transfers. PVM does not support such an abstraction, leaving the programmer to manually arrange tasks into groups with the desired communication organization.

## 5. Fault Tolerance

Fault tolerance is a critical issue for any large scale scientific computer application. Long-running simulations, which can take days or even weeks to execute, must be given some means to gracefully handle faults in the system or the application tasks. Without fault detection and recovery it is unlikely that such applications will ever complete. For example, consider a large simulation running on dozens of workstations. If one of those many workstations should crash or be rebooted, then tasks critical to the application might disappear. Additionally, if the application hangs or fails, it may not be immediately obvious to the user. Many hours could be wasted before it is discovered that something has gone wrong. Further, there are several types of applications that explicitly require a fault-tolerant execution environment, due to safety or level of service requirements. In any case, it is essential that there be some well-defined scheme for identifying system and application faults and automatically responding to them, or at least providing timely notification to the user in the event of failure.

PVM has supported a basic fault notification scheme for some time. Under the control of the user, tasks can register with PVM to be "notified" when the status of the virtual machine changes or when a task fails. This notification comes in the form of special event messages that contain information about the particular event. A task can "post" a notify for any of the tasks from which it expects to receive a message. In this scenario, if a task dies, the receiving task will get a notify message in place of any expected message. The notify message allows the task an opportunity to respond to the fault without hanging or failing.

Similarly, if a specific host like an I/O server is critical to the application, then the application tasks can post notifies for that host. The tasks will then be informed if that server exits the virtual machine, and they can allocate a new I/O server. This type of virtual machine notification is also useful in controlling computing resources. When a host exits from the virtual machine, tasks can utilize the notify messages to reconfigure themselves to the remaining resources. When a new host computer is added to the virtual machine, tasks can be notified of this as well. This information can be used to redistribute load or expand the computation to utilize the new resource. Several systems have been designed specifically for this purpose, including the WoDi system [7] which uses Condor [6] on top of PVM.

There are several important issues to consider when providing a fault notification scheme. For example, a task might request notification of an event after it has already occurred. PVM immediately generates a notify message in response to any such "after-the-fact" request. For example, if a "task exit" notification request is posted for a task that has already exited, a notify message is immediately returned. Similarly, if a "host exit" request is posted for a host that is no longer part of the virtual machine, a notify message is immediately returned. It is possible for a "host add" notification request to occur simultaneously with the addition of a new host into the virtual machine. To alleviate this race condition, the user must poll the virtual machine after the notify request to obtain the complete virtual machine configuration. Subsequently, PVM can then reliably deliver any new "host add" notifies.

The current MPI standard does not include any mechanisms for fault tolerance, although the upcoming MPI-2 standard will include a notify scheme similar to PVM's. The problem with the MPI-1 model in terms of fault tolerance is that the tasks and hosts are considered to be static. An MPI-1 application must be started *en masse* as a single group of executing tasks. If a task or computing resource should fail, the entire MPI-1 application must fail. This is certainly effective in terms of preventing leftover or hung tasks. However, there is no way for an MPI program to gracefully handle a fault, let alone recover automatically.

The reasons for the static nature of MPI are based on performance as well as convenience. Because all MPI tasks are always present, there is no need for any time-consuming lookups for group membership or name service. Each task already knows about every other task, and all communications can be made without the explicit need for a special daemon. Because all potential communication paths are known at startup, messages can also, where possible, be directly routed over custom task-to-task channels.

MPI-2 will include a specification for spawning new processes. This expands the capabilities of the original static MPI-1. New processes can be created dynamically, but MPI-2 still has no mechanism to recover from the spontaneous loss of a process. One of the fundamental problems that keeps MPI from being fault tolerant is the synchronous way that communicators are created and freed. In the next section we compare the way MPI and PVM handle communicators and communication context in general.

## 6. Context for Safe Communication

The most important new concept introduced by MPI is the *communicator*. The communicator can be thought of as a binding of a communication context to a group of processes. Having a communication context allows library packages written in message passing systems to protect or mark their messages so that they are not received (incorrectly) by the user's code. Message tag and sender ID is not enough to safely distinguish library messages from user messages. Figure 1 illustrates the fundamental problem. In this figure two identical worker tasks are calling a library routine that also performs message passing. The library and user's code have both chosen the same tag to mark a message. Without context, messages are received in the wrong order. To solve this problem, a third tag that is assigned by the operating system is needed to distinguish user messages from library messages. Upon entrance to a library routine, for example, the software would determine this third tag and use it for all communications within the library. The remainder of this section will compare and contrast the specification of context in MPI and PVM version 3.4. Context primitives are a new feature in PVM 3.4.

Context is assigned by the operating environment and cannot be wildcarded by a user program. Two important issues are how this "magic" tag is derived and how the tag is distributed to all processes that need to use it for communication.

MPI couples the concepts of context and a group of processes into a communicator. When a program starts, all tasks are given a "world" communicator and a (static) listing of all the tasks that started together. When a new group (context) is needed, the program makes a synchronizing call to derive the new context from an existing one (intra-communication). The derivation of context becomes a synchronous operation across all the processes that are forming a new communicator. This has several advantages: no servers are required to dispense a context, instead processes need only decide among themselves on a mutually exclusive safe context tag; all context state is dissolved (and hence re-usable) when one or more of the processes terminates; and, derivation and distribution of context are always performed in a single call. However, in MPI it is possible (and in fact common in existing implementations) for two independent groups of processes to use the same context tag. The MPI forum decided it was too difficult and expensive to generate a unique context tag. This means that it is unsafe for two groups to send messages to each other. To solve this problem, MPI introduces an inter-communicator which allows two groups of processes to agree upon a safe communication context. Collective operations such as broadcast are not supported over inter-communicators, but this is be-

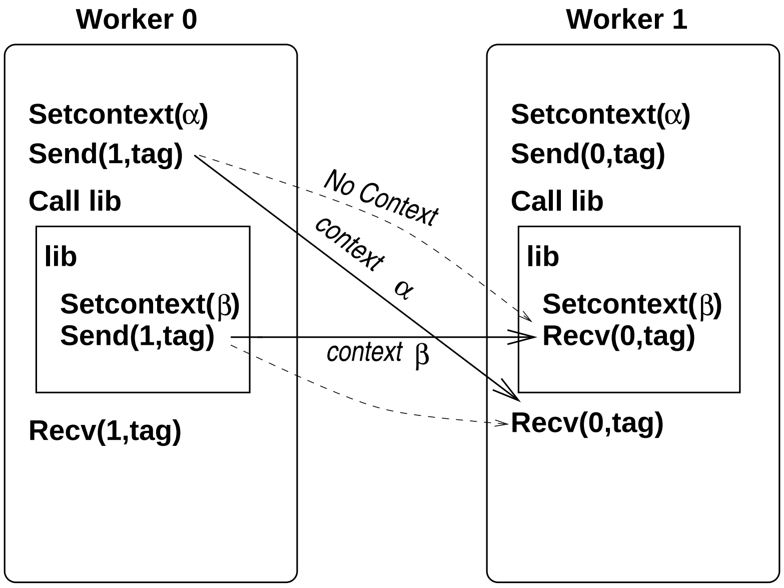


Figure 1: Messages sent without context are erroneously received

ing discussed for inclusion in MPI-2. Having two types of communicators and having to convert between them is sometimes confusing, always messy, but necessary given MPI's mandate not to require a server or daemon process. The static nature of communicators makes it cumbersome to spawn new tasks and enable communication between old and new tasks. Finally, when a task fails, the world communicator becomes invalid. Because all intra-communicators are derived from this now invalid construction, the behavior of the program becomes undefined (and implementation dependent).

Since PVM already has a set of daemon processes maintaining the virtual machine, it can use these to create a system-wide unique context tag, which leads to a simpler and more general context model. PVM 3.4 views context as merely another tag (without wildcarding). Primitives are provided for tasks to request a new unique context tag. The program may use and distribute this context in exactly the same way that MPI does. For example, in PVM 3.4 there is a synchronous group formation operation that allocates and distributes a new context to all members. This is analogous to the MPI group formation. But in PVM the different groups are each guaranteed a unique context, removing the need for separate intra- and inter-communication constructs. As long as a process knows (or can find out) the context of a group, it can communicate with that group.

Other advantages to the PVM context scheme are that new processes can use existing contexts to start communicating with an existing group. This is especially useful for fault-tolerant programs that want to replace failed tasks.

In contrast, if a task fails in MPI, the result is undefined and usually leads to system shutdown. The generality of the PVM approach, however, results in some thorny issues involving when to reclaim context values for reuse. We consider an existing communicator to be *corrupted* if one or more member processes fail before the communicator has been collectively freed. There is no one "correct" answer on how to deal with corrupted communicators. Automatic recycling may cause errors in fault-tolerant programs. No recycling may exhaust system resources.

MPI-1 side steps this issue by requiring tasks to never fail. MPI-2 will have to address this problem since it allows tasks to spawn new tasks and processes may be notified about other failed processes. The fundamental problem is that the MPI_COMM_WORLD communicator, from which most intra-communicators are derived, becomes corrupted. This leads to undefined consequences.

To maintain backwards compatibility with existing PVM codes, a context tag has not been explicitly added to the argument lists of pvm_send and pvm_recv. Instead, a PVM task has an *operating* context that can be queried and set by the program. Sends and receives are carried out in the current operational context.

PVM 3.4 has the concept of a base context. All tasks will know the base context and will be able to use it for communication even in the event of task failure. In PVM, a spawned task will inherit the current operational context of its parent. If the task has no parent (i.e., the task was started from the command line), then it operates in the base context. Inheritance provides an easy way for a master to encapsulate the messages of several parallel worker groups.

PVM 3.4 will add some MPI-like communicator constructions, which are shown in Table 1. The collective call pvm_staticgroup() takes a list of tids and returns a group communicator id. This allows PVM programs to arbitrarily construct groups of processes with context. Group send and receive functions will be added that take a group id and a rank as arguments. These functions are "syntactic sugars" that handle the mappings of group rank to tids and the proper management of the operating context.

Not having a context argument in pvm_send points out an advantage MPI has over PVM in general. MPI was designed to be thread-safe, that is, there is no hidden state that could complicate writing a multi-threaded application. PVM has hidden state in both the active context and the active message buffer. This requires programmers to be very careful when writing multi-threaded PVM applications.

## 7. Name Service

It is often desirable for two programs to start independently and discover information about each other. A common mechanism is for each of the programs to key on a "well-known name" to look up information in a database.

| function | PVM | MPI-1 |
|---|---|---|
| communicator creation | pvm_staticgroup | MPI_COMM_CREATE MPI_INTERCOMM_CREATE |
| communicator destruction | pvm_lvgroup | MPI_COMM_FREE |
| new context | pvm_newcontext | MPI_COMM_DUP |
| inter-communication | no restrictions | some restrictions |
| support fault tolerance | yes | no |

Table 1: Similarities between manipulating communicators in PVM and MPI

A program that returns information about a requested name is called a *name server*.

PVM is completely dynamic. Hosts may be added to and deleted from the virtual machine. Processes may start, run to completion and then exit. The dynamic nature of PVM makes name service very useful and convenient. In PVM 3.4, the distributed set of PVM daemons have added functionality to allow them to perform name server functions.

In comparison, MPI-1 supplies no functionality that requires a name server. MPI-2 proposes to add functions to allow independent groups of processes to synchronize and create an inter-communicator between them. The functions are being defined so as not to mandate the use of a name server, allowing implementations the freedom to use existing server software.

In PVM 3.4, there is a general name service. A PVM task or PVM daemon can construct an arbitrary message and 'put' this message in the name server with an associated key. The key is a user defined string. Tasks that look up a name are sent the stored message. This sort of name service is a very general mechanism. The message could, for example, contain a group of task ID's and an associated context, the location of a particular server, or simply initialization data.

The insertion allows a task to specify the owner of the named message. The owner may be any process, including a pvmd. A message is deleted when an owner (or the pvmd) explictly calls pvm_del, when the owner exits the virtual machine, or when the entire virtual machine is reset. Two tasks may not insert the same name since only the first insertion will succeed. The second insertion returns an error. This has the advantage that third party modules, such as visualizers, may start up, query the name server, and insure that they are unique across the entire virtual machine.

There are three basic calls for the PVM name server:

| | |
|---|---|
| pvm_putmsg() | Insert (key,message) pair into name server and specify owner |
| pvm_getmsg() | Return inserted message (if any) from key |
| pvm_delmsg() | Owner may delete a message from name server |

Table 2: Routines that support name service

# 8. Message Handlers

User-level message handlers provide an extensible mechanism for building event-driven codes that easily co-exist with traditional messaging. Both PVM 3.4 and MPI-2 will have user-level message handlers. A program may register a handler function so that when a specified message arrives at a task, the function is executed. Message handlers are therefore very similar to active messages with the caveat in PVM that they cannot interrupt a program while it is operating outside of the PVM library. There will be two PVM interface calls for message handlers:

| | |
|---|---|
| pvm_addmh() | Add a message handler function matching (src,tag,context) |
| pvm_delmh() | Remove a previously defined message handler |

Table 3: Routines that support message handlers

PVM has always had message handlers for system use. For example, when a direct connection is requested, a message is sent from the requester to the receiver. This request message is intercepted by the pvm library software before it can be received by the user's code. The handler opens a socket and replies to the requestor.

There are many possibilities for using message handlers. The following example code segment shows a handler that returns the contents of a local counter:

```
static int counter;
void show_count()
{
    sbuf = pvm_setsbuf(0);         /* remember the current send buf */
    mysbuf = pvm_initsend(PvmDataDefault);
    pvm_pkint(&counter,1,1);
    pvm_send(src,tag);
    pvm_freebuf(mysbuf);
    pvm_setsbuf(sbuf);            /* restore the send buf */
}
```

```
main()
{
    pvm_addmh(show_count);
    while(1) {
        counter ++;
        (do message passing)
    }
}
```

The message handler can be called upon entry to any PVM library call. It is clear from the above code segment that the user must be very careful to save and restore any modified PVM state.

## 9. Future Research: PVMPI

The Universitiy of Tennessee and Oak Ridge National Laboratory have recently begun investigating the feasibility of merging features of PVM and MPI. The project is called PVMPI [1] and involves creating a programming environment that allows access to the virtual machine features of PVM and the message passing features of MPI.

PVMPI would perform three symbiotic functions: It would use vendor implementations of MPI when available on multiprocessors. It would allow applications to access PVM's virtual machine resource control and fault tolerance. It would transparently use PVM's network communication to transfer data between different vendor's MPI implementations allowing them to interoperate within the larger virtual machine.

## 10. Conclusion

The recent publicity surrounding MPI has caused programmers to wonder if they should use the existing de facto standard, PVM, or whether they should shift their codes to the MPI standard. In this paper we compared the features of the two APIs and pointed out situations where one is better suited than the other.

If an application is going to be developed and executed on a single MPP, then MPI has the advantage of expected higher communication performance. The application would be portable to other vendor's MPP so it would not need to be tied to a particular vendor. MPI has a much richer set of communication functions so MPI is favored when an application is structured to exploit special communication modes not available in PVM. The most often cited example is the non-blocking send.

Some sacrifices have been made in the MPI specification in order to be able to produce high communication performance. Two of the most notable are the lack of interoperability between any of the MPI implementations, that is, one vendor's MPI cannot send a messages to another vendor's MPI. The second is the lack of ability to write fault tolerant applications in MPI. The MPI specification states that the only thing that is guaranteed after an MPI error is the ability to exit the program.

Because PVM is built around the concept of a virtual machine, PVM has the advantage when the application is going to run over a networked collection of hosts, particularly if the hosts are heterogeneous. PVM contains resource management and process control functions that are important for creating portable applications that run on clusters of workstations and MPP.

The larger the cluster of hosts, the more important PVM's fault tolerant features become. The ability to write long running PVM applications that can continue even when hosts or tasks fail, or loads change dynamically due to outside influence, is quite important to heterogeneous distributed computing.

Programmers should evaluate the functional requirements and running environment of their application and choose the API that has the features they need.

# References


[1] Graham E. Fagg and Jack J. Dongarra. PVMPI: An integration of the PVM and MPI systems. *Calculateurs Paralleles*, 2, 1996.

[2] MPI Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Application*, 8 (3/4):165 – 416, 1994.

[3] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine*. MIT press, 1994.

[4] G. A. Geist and V. S. Sunderam. Network-based concurrent computing on the PVM system. *Concurrency: Practice & Experience*, 4 (4):293 – 311, 1992.

[5] Ken Koch and Harvey Wasserman. A message passing algorithm for Sn transport. In *Proceedings of 1996 PVM User Group meeting*, 1996. http://bay.lanl.gov/pvmug96.

[6] Jim Pruyne and Miron Livny. Providing resource management services to parallel applications. In *Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing*, 1994. http://www.cs.wisc.edu/condor/publications.html.

[7] Jim Pruyne and Miron Livny. Parallel processing on dynamic resources with CARMI. In *Proceedings of IPPS'95*, 1995. http://www.cs.wisc.edu/condor/publications.html.

[8] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice & Experience*, 2 (4), 1990.

[9] BBN ToolWorks. Totalview parallel debugger. http://www.bbn.com:80/tv.

[10] S. VanderWiel, D. Nathanson, and D. Lilja. Performance and program complexity in contemporary network-based parallel computing systems. Technical Report HPPC-96-02, University of Minnesota, 1996.