PVMPI: An Integration of the PVM and MPI Systems

Graham E. Fagg *   Jack J. Dongarra†

April 12, 1996

Abstract

We discuss the use of PVM as a system for controlling the execution of MPI applications, by allowing the user access to both the MPI API and an enhanced set of the PVM API. The intention is to give the user community flexible control over MPI applications using a system that is both portable and familiar—without having to wait for new MPI-2 systems to be developed. Our system, called PVMPI, uses the already proven and widely ported MPI message-passing system *within PVM* to to enable interoperation with different implementations executing on distributed hardware. PVMPI also takes advantage of contexts under PVM3.4 to provide more security. Additional benefits will be available to those who currently already use resource managers that interface to PVM, in that PVMPI can control MPI applications.

1 Introduction

PVM is one of a number of parallel distributed computing environments (DCEs) [16] that were introduced to assist users wishing to create portable parallel applications [19]. The system has been in use since 1992 [1] and has grown in popularity, leading to a large body of knowledge and a substantial quantity of legacy code accounting for many man-years of development.

For the past several years, standardization efforts have attempted to address many of the deficiencies of the different DCEs and introduce a single stable system for message passing. These efforts culminated in the first Message Passing Interface (MPI) standard, introduced in June 1994 [13]. Within a year, several different implementations of MPI were available, including both commercial and public systems.

One of MPI's prime goals was to produce a system that would allow manufacturers of high-performance massively parallel processing (MPPs) computers to provide highly optimized and efficient implementations. In contrast, PVM was designed primarily for networks of workstations, with the goal of portability, gained at the sacrifice of optimal performance. PVM has been ported successfully to many MPPs by its developers and by vendors, and several enhancements—including in-place data packing and pack-send extensions—have been implemented with much success [3]. Nevertheless, PVM's inherent message structure has limited overall performance when compared with that of native communications systems.

*Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301

†Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301 and Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831-6367

Thus, PVM has many features required for operation on a distributed system consisting of many (possibly nonhomogeneous) nodes with reliable, but not necessarily optimal, performance. MPI, on the other hand, provides high-performance communication and a nonflexible static process model.

The aim of this work is to interface the flexible process and virtual machine control from the PVM system with the enhanced communication system of several MPI implementations. The need for such a system was clearly identified by the first MPI forum and motivated the current round of discussions by the MPI-2 forum. Indeed, MPI-2–style tools have been promised by several MPI implementors; examples include MPIX [17] and LAM MPI-6.0 [2].

In this paper we compare the PVM and MPI systems, focusing in particular on machine definition, process control, and message-passing implementation. Then we consider how these systems can interoperate, and we address such issues as language binding and interfaces.

2 The Dynamic World of PVM

PVM (Parallel Virtual Machine) relies on the idea that the system the software runs upon is not fixed, but is dynamic. This dynamic approach forces application developers to avoid making assumptions about the underlying system. Such an approach offers two major advantages over a static resource world:

1. A truly portable API, supporting over twenty different platform types simultaneously.
2. A flexible system that can be easily made fault tolerant in the event of node loss, as well as being able to take advantage of addition nodes during run time.

Although to a user or application developer only the API is of utmost importance, we briefly discuss here the internal workings of PVM in order to illustrate the difference between it and related systems.

First, PVM is generally considered as a basic message-passing system built upon a system of generic daemons. Specifically, PVM API calls cause the user's application to coordinate with a daemon *pvmd* to perform some operation (e.g., send a message or start some application). The daemons themselves define the virtual machine and provide the same consistent run-time environment across diverse platforms, thereby allowing for a single API.

When running on a netowrk of workstations (NOW), each host becomes a member of the virtual machine by running its own daemon. Applications become PVM applications by coordinating with these daemons via sockets and/or pipes. Thus, applications can become enrolled into PVM and then be controlled by it even if they were not started by it. Applications can also join and leave as many times as they wish, allowing them to live through several different virtual machines.

Shared-memory processor systems (SMPs), such as Sun MPs and SGI, require only a single daemon per host regardless of the number of nodes they contain. This method of process control is possible because they use the same Unix mechanisms as general workstations to initiate and signal processes. Dedicated MPPs, on the other hand, do not always offer such a consistent method of initialing and controlling processes. Like SMPs, they usually have only a single daemon running on a front end or service node. These daemons provide the same or similar facilities and services as conventional daemons except they may use different specialized system calls to interface with the MPP's nodes. Restrictions placed upon PVM by these run-time systems may affect the API functionality offered. or example, IBM's PVMe allows only an SPMD model to be used, and the Meiko CS2 allows only barrier operations across the whole application.

An example of a system that can appear as either a NOW or an MPP is the IBM SP2. In its NOW form it runs a daemon on each node as a conventional cluster of Unix workstations. As an MPP, it runs only a single daemon and uses the local partition management software to control resource allocation.

2.1 Virtual Machine Definition

The virtual machine is defined by the number and location of the running daemons. Although the number of hosts can be indicated by a fixed list at start-up time, there exists only a single point of failure, the first master daemon to start. All other hosts can join, leave, or fail without affecting the rest of the virtual machine.

PVM API functions allow the user to

- add or delete hosts,
- check that a host is responding,
- be notified by a user-level message that a host has been deleted (intentionally or not) or has been added, and
- shut down the entire virtual machine, killing attached processes and daemons.

2.2 Process Control

PVM API functions provide the ability to

- join or leave the virtual machine;
- start new processes by using a number of different selection criteria, including external schedulers and resource managers;
- kill a process;
- send a signal to a process;
- test to check that it is responding; and
- notify an arbitrary process if another disconnects from the PVM system.

2.3 Message Passing

Two types of message passing exist in PVM: (1) internal (between daemons and other daemons or user tasks), and (2) user (between two or more user processes enrolled into PVM).

User messages are identified by source address and a single user-controlled tag, which can be used by a task to filter incoming messages from multiple destinations. The lack of contexts prior to PVM 3.4 made safe message passing for libraries extremely difficult. Although the user could use a set of reserved tags, no guarantee existed for these tags being unique across the virtual machine, since they were user chosen and not system allocated.

2.4 Resource Management

PVM's comprehensive array of API routines allows the user the same level of control over the virtual machine as the system has. This flexibility has encouraged many projects to use PVM in different distributed computing environments [14] such as Mist [12], dedicated schedulers [15], load balancers, and process migration tools [4, 18].

2.5 PVM Group Services

PVM provides the ability for processes to form into groups identified by a character string name, which is held in a single central database process called the PVM Group Server *PVMGS*. Processes can join and leave any number of groups at any time, making membership completely dynamic. Processes are allocated instance numbers when they join, in the order that they join a group. The first join operation creates the group, and the group is destroyed when the membership falls to zero (i.e., no empty groups), although groups may have gaps in their membership as processes leave out of order.

The group service provides a limited number of collective operations such as barrier and reduce. Also provided is a broadcast operation that allows messages to be sent to all members of a group (unless the sender is one). However, there are no point-to-point operations on a group. Thus, the user must explicitly look up process addresses and use the normal point-to-point send and receive primitives.

Until recently, the group database was centrally stored and thus had to be accessed before each group operation took place, even if the group had not altered. This strategy led to serious degradation in potential performance. Later versions of PVM 3.3 enabled the groups to be frozen and their details to be cached locally. In some cases, full dynamic group caching has also been developed [8].

3 The Static World of MPI

As previously stated, many systems provide only a static process control model. The nodes in such a model may be fixed at compile, download, or spawn time; and once started, an application cannot usually change size or migrate during its execution. Failure of a single program module causing the entire application to fail by invalidating its message communicators.

Because of the wide range of possible initialization options, the MPI forum decided against standardizing process control. This decision had several advantages:

- The process model was easy to reason about. A fixed number of processes existed: either all processes existed or none.
- Collective operations could be optimized, since the members taking part were known beforehand and were not subject to change.
- Contexts or application tags could be implemented efficiently.

3.1 MPI Process Control

Although the MPI standard does not state how processes are started, it does state how and in which order processes become MPI processes. All MPI processes join the MPI system by calling MPI_Init and leave by calling MPI_Finalize. Processes calling MPI_Init twice may have an undefined behavior.

3.2 Contexts, Process Groups, and Communicators

Context is a system-defined tag that can be used to differentiate messages from one another. Contexts generally are used by different layers in a library to eliminate possible interference between these layers. In PVM, for example, any task can send a message to any other task, whether the receiving task wishes to interact with the sender or not (as in the case of two separate applications). In MPI, on the other hand, the two applications have two separate message universes or contexts, which render this potential mistake impossible.

Processes in MPI are arranged in rank order, from 0 to N-1, where N is the number of processes in a group. These process groups define the scope for all collective operations within that group.

The process group and context, together with other information about topologies and local attributes, constitute a communicator. All communications can operate only within a communicator. Thus, this strategy gives rise to a high level of protection against rogue messages.

3.3 Static Separate Worlds

Once all the expected processes have joined the system, a common communicator is created by the system for them. This communicator, referred to as MPI_COMM_WORLD, will allow all processes to communicate to all others in their "world." From this communicator, subset communicators can be created and duplicated for the different modules in an application by using different (possibly overlapping) groups of processes.

Communications between processes within the same communicator or group are referred to as *intracommunicator communications*. Communications between different groups are *intercommunicator communications.* The formation of an intercommunicator requires two separate (nonoverlapping) groups and a common communicator between the leaders of each group, as shown in Figure 1.

With the current standard MPI-1 implementation it is impossible to create an intercommunicator between two separately initiated MPI applications. Each application has its own MPI_COMM_WORLD, with no existing communicator bridging the gap between them (see Figure 2).

All internal details are hidden from the user. MPI communicators have relevance only within a particular run-time instance. Moreover, this strategy excludes different MPI implementations from interoperating. In summary, current MPI applications are static and isolated. and communication between them will probably not be possible via message passing, but by other mediums such as cross-mounted file systems.

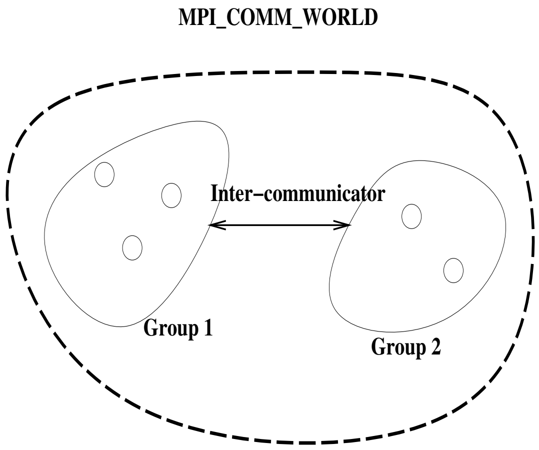

Figure 1: Intercommunicator formed inside a single MPI_COMM_WORLD

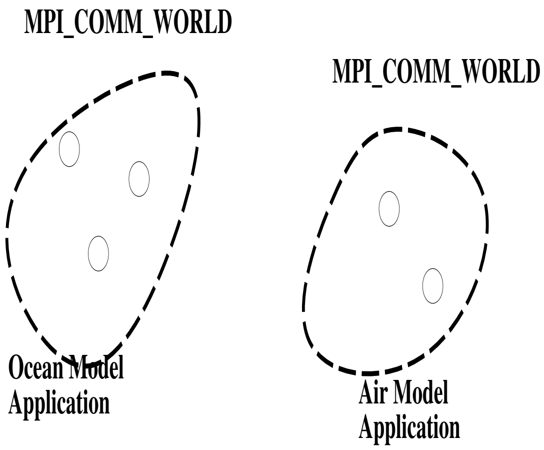

Figure 2: Separate applications that are unable to create an intercommunicator because they lack any overlapping communicator

4 Related Work

Although several MPI implementations are built upon established message-passing libraries such as Chameleon-based MPICH [6] and the LAM system [2], Unify [5] from Mississippi State University is the closest related project in terms of dual APIs, with LAM 6.0 being closest in dynamic support.

The MPIX project also from Mississippi State University has some bearing on this research effort in that it extends the capability of current MPI intercommunicators to allow them to be used in collective operations instead of only in point-to-point operations. Also supported are overlapping groups, which currently are not allowed in MPI.

4.1 Unify

The Unify system was originally proposed to *unify* or mate together the PVM and new MPI APIs. The intention was to enable users to take current PVM applications and slowly migrate toward complete MPI applications, without having to make the complete conceptual jump from one system to the other.

The project, which was a masters degree project, never reached full maturity in that many MPI features (such as virtual topologies, profiling, attribute caching, and intercommunicators) were not implemented. Although all the MPI intracommunicator point-to-point and collective operations were included, Unify failed to exploit PVM's dynamic spawning capability (hence no need for intercommunicators) and forced the user to spawn a fixed number of master-slave SPMD processes from the command line. More specifically, the start-up sequence consisted of a process that checked to see whether it was a master by the existence of a parent process and then spawned N-1 copies of itself. If a parent existed, the process was assumed to be a slave and would block on a receive, awaiting a TID list so that it could build its MPI_COMM_WORLD, MPI_COMM_SIZE and MPI_COMM_RANK values. Thus, a Unify application could not be started by any other PVM process (including the console). Moreover, it could not use PVM spawn to start other MPI applications, since their MPI_Init calls would wait for never-arriving start-up messages.

Unify did address the difficulty of mapping identifiers between the PVM and MPI domains, where each system used a different scheme; PVM using a 32-bit integer and MPI using a handle to an opaque internal structure together with a rank inside that structure. Unify provided only two new additional calls: one from MPI to PVM tid, and vice versa (without restrictions, since all the tasks were running within both the PVM and the MPI environments).

5 Interoperation Requirements and Membership Rules

For PVM to interconnect any two groups of processes and allow them to communicate, at least one process in each group must be enrolled into PVM. Processes can become enrolled into PVM by being started by PVM (i.e., implicitly) or by calling a PVM library function (i.e., explicitly). An implicitly started applications may also wish to remain independent, for example, when PVM is used only as a start-up facility. Thus, the system must make sure that even if PVM fails pathologically, it must not be able to interfere with the application's life cycle in any way.

The scope of any communications will depend upon the completeness of membership, that is, fully connected to both systems or partially connected. If full connectivity is not possible, intercommunicator operations could use point-to-point only communications between subsets of nodes. Alternatively, by using extra calls, such operations could use the connected nodes as relays to complete connections.

Another factor in membership is its duration. MPI applications may interact with each other only in a server-client behavior pattern, as in the case of computational steering and visualization, and may not wish to be part of the PVM system continuously. Hence the PVMPI membership is required to be dynamic, as with the current PVM group services, although many PVMPI operations may be required to be blocking and collective to aid correctness, as with current MPI practice.

6 Prototype Systems

A prototype system has been developed to ease the interconnection of MPI and PVM. Four separate issues have been addressed:

1. mapping identifiers or managing MPI and PVM IDs,

2. start-up facilities and process management,
3. MPI-style PVM message passing and collective operations, and
4. improved security and performance with attribute locking.

6.1 Mapping Identifiers

Processes in an MPI application are identified by referencing a tuple pair such as {process group, rank} or {communicator, rank}. PVM also has this capability when using the group library, in the form of {group name, instance}.

In the simplest case, where all the processes in an MPI application group have access to PVM, a single pair of calls can be used to register a process group with the current PVM group server. The functions are available in both C and Fortran bindings:

```
info = pvmpi_register(char *group, MPI_Comm comm, int *options);

info = pvmpi_leave(char *group);

call pvmpifregister( group, comm, options )

call pvmpifleave ( group )
```

Both functions are collective: all processes in the MPI communicator have to call them together.

The pvmpi_leave command is used to clean up MPI data structures and to leave the PVM system in an orderly way if required.

Processes can register in multiple groups, although currently separate applications cannot register into a single group with this call. The register call takes each member of the context and makes it join a named PVM group so that its instance number within that group matches its MPI rank. Since any two MPI applications may be executing on different systems using different implementations of MPI (or even different instances of the same version), the communicator usually has no meaning outside of any application callable library. The PVM group server, however, can be used to resolve identity when the groups names are unique.

Once the application has registered, an external process can now access any registered process by using that processes group name and instance via the library calls pvm_gettid and pvm_getinst. When the groups have been fully formed without any errors occurring, they are frozen and all their details are cached locally so that there are very few system over-heads for accessing them using the group library.

Figure 3 shows the previous example applications using the register group call, and figure 4 shows the new groups communicating using conventional PVM calls.

Client-server interactions often require waiting for applications partners to start. To handle this situation, an additional blocking call has been provided that waits until a group has completely registered before returning its size and caching its addresses locally:

```
groupsize = pvmpi_waitfor (char* group);

call pvmpifwaitfor ( group )
```

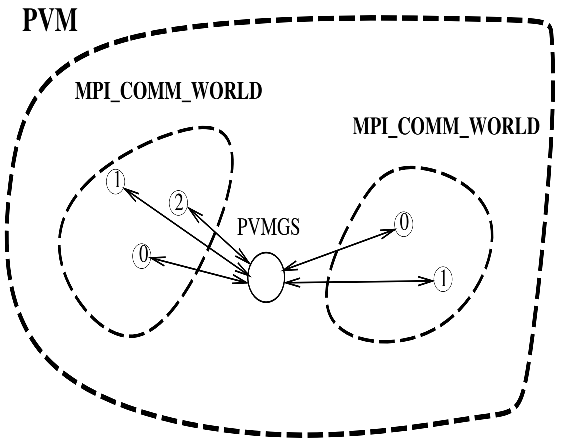

Figure 3: Two separate MPI applications register their process groups by using pvmpi_register().

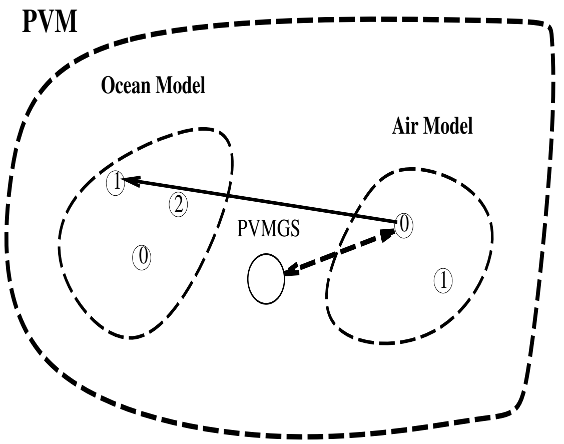

Figure 4: The zeroth rank "Air Model" process sending a PVM message to the first rank "Ocean Model" process with pvm_send(pvm_gettid("Ocean_Model",1), tag)

This routine not only removes the need for a user to poll the group server, but also helps prevent races caused by the dynamic nature of PVM groups [8]. If two applications are started separately, they may not have fixed sizes, and so they may not know when it was safe to start communicating with each other without additional handshaking. This routine eliminates the need for such additional handshaking.

6.2 Start-up Facilities and Process Management

The spawning of MPI jobs requires different procedures depending upon the target system and the MPI implementation. The situation is complicated by the desire to avoid adding many spawn calls (the current intention of the MPI-2 forum). Instead, a number of different resource managers and MPI implementation specific taskers have been developed. This work has been the impetus behind a simplification of the current resource management hooks so that expansion of the PVM system itself is more modular.

Three basic schemes are available:

1. An application schemer is created, and the system forks the required version of MPIRUN to start the MPI job.

2. Taskers intercept the calls and modify either the arguments passed to the new processes or the working environment.

3. Current default: Processes are started as normal Unix tasks.

The first method is being used on various MPP versions of PVM, such as for the SP2 when using the SP2MPI variate. In these cases the resource manage and taskers must work closely together to ensure that the created groups of processes have the "correct" PVM parent ID. This method is also used for MPIF applications and for MPICH and LAM applications depending on circumstances.

The second method is used for MPICH applications running under the ch_p4 device on workstation clusters. This method currently alters the argument list passed to the processes. When MPI-2 eliminates the mandatory passing of {argc,argv} to MPI_Init, this will be changed to alter the environment as required.

The third method is applicable to LAM processes when the user requests a single process per LAM/PVM node.

These systems require the user to adhere to some superficial constraints, such as placing MPI executables in user-configurable directories so that their nature can be determined from their location. The declaration of available nodes in the case of LAM5.X and MPICH is also required before spawn time. Since LAM 6.0 can alter its virtual machine, this has to be polled at spawn time by a specialized tasker running on one of its nodes.

The spawn command has not altered, although when interfacing to a resource manager, it is allowed to be called with one of the following additional flags—PVMPI_LAM or PVMPI_MPICH—in place of the current spawn flag options. If a specialized tasker is used, spawning is identical to spawning on a MPP front-end or service node:

```
pvm_spawn( "MPI_APP",..,PvmTaskHost,"Host_in_MPI_system", N, ..)
```

6.2.1 SP2 Process Spawning

The SP2 version of PVM that uses MPI for internal communications appeared initially not to require any alteration. Unfortunately, when required to spawn N processes, it spawned an extra process to manage communication with the daemon, effectively allowing true nonblocking communication between on and off machine nodes, as shown in Figure 5 for a four-task application. In other words, MPI applications have one extra process that they cannot communicate with, because it is dedicated to relaying messages for the PVM system.

The library also had other shortcomings in its ability to handle re-entrance of MPI_Init. In particular, its default failure mode was pathological, and it did not use a private communicator internally, but instead used MPI_Comm_World. Two separate systems [7] are currently being evaluated that resolve these problems:

1. A modified SP2MPI port that creates the required number of tasks, each of which individually opens a socket to the spawning PVM daemon for out of application communication. This

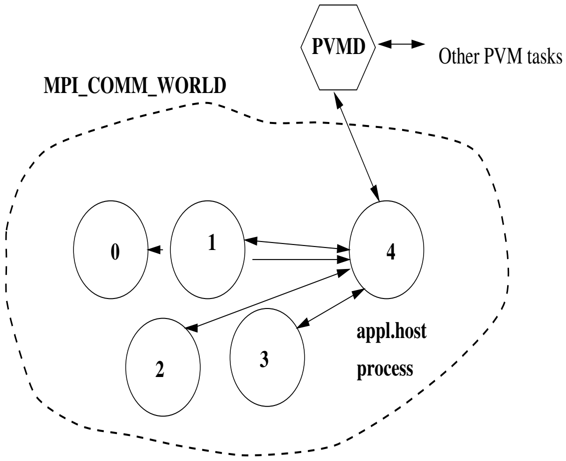

Figure 5: SP2MPI PVM using an application host task to manage nonapplication message routing

version works for both PVMPI and conventional PVM applications and may replace the currently used method.

2. An SP2 viewed as a cluster of RS/6000 workstations, but with the application spawned as in the SP2MPI version using the IBM POE[11] called from a special tasker. The tasks individually connect to their local host daemon, after an extra layer of handshaking that correctly sets up their task data structures (including the parent task identity).

6.3 MPI-style Message Passing and Collective Operations

Using two different styles of API for message passing as opposed to process control may in itself cause difficulties for users, especially if they have never used PVM or the MPI buffered pack routines before. Thus, some basic send and receive operations are provided in a similar form to the original MPI buffered operations. For example, the routines

```
pvmpi_send(void* buf, int count, MPI_Datatype dtype, int destination, int tag,
char* group)
pvmpi_recv(void* buf, int count, MPI_Datatype dtype, int destination, int tag,
char* group)
```

are used in the same way as the current MPI point-to-point operations except that a group name is given instead of a communicator handle. They support basic continuous data types with more advanced derived data types (see the PVM CCL project [9]).

The need for collective operations across communicators has been identified by other research groups and has led to an experimental library, based upon MPICH, called MPIX (MPI eXtensions) [17]. The library allows many of the current intracommunicator operations to work across intercommunicators such as *All Gather* and *All to all*.

The current PVM group services, based upon the pvm_bcast function, be used to link different implementations of MPI. Again, PVMPI operations to ease the use of groups can be created and are currently being investigated. One operation of particular interest is an intercommunicator *sendrecv* call. This call assists the synchronization of two independent applications and allows them to exchange data in a convenient way that matches many domain decomposition models (see Figure 6).

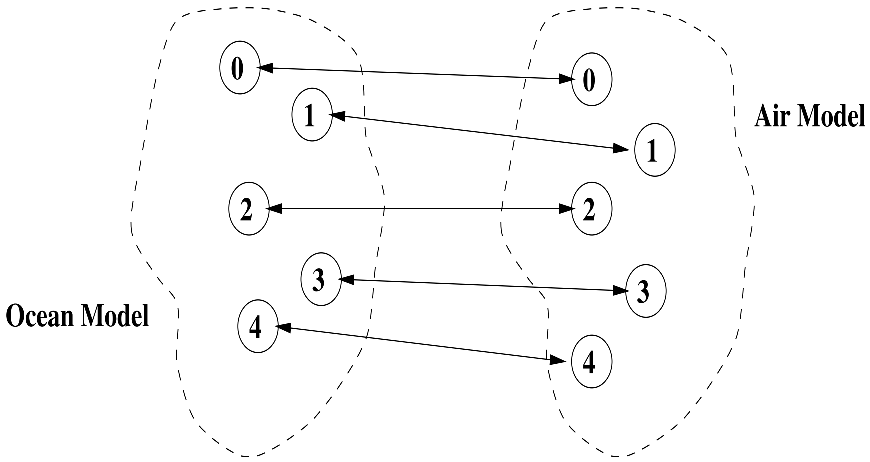

Figure 6: Passing boundary values between two separately initiated applications using pvmpi_sendrecv

6.4 Improved Security and Performance with Attribute Locking

PVM 3.4 [10] includes many enhancements such as contexts and a mailbox-style user-accessible database. These features can be used to add a level of protection on a par with that of MPI. Thus, by enrolling in PVMPI, an MPI application can still be protected from rogue messages. The second implementation of the PVMPI system uses some of the PVM 3.4 features to stores group attributes in the mailbox. Attributes include group size, node architectures, context tag, and access permissions, which are set by using the options entry in the pvmpi_register call. Once a process has registed under PVM 3.4, it is issued with a context for its group. External processes cannot access details about that group, such as size or membership, unless it matches the permission criteria set during the registration.

The permission options for registration include the following:

- PVMPI_ANY: Any process can look up a groups details and attributes. This is the default for the PVM 3.3 group services.

- PVMPI_SIBLIN: Any process that shares a common parent can access details. The number of levels up the process tree searched can be varied.

- PVMPI_CHILD: Only this process's children can find details about it.

- PVMPI_PRIVATE: Access to process's identity and context for any external processes is disallowed.

Processes may change their access permission at any time, noting that this is a collective operation across the entire group.

Once a group has registered, its context is set and stored in the system mailbox. If a process attempts to communicate with these processes, it must obtain the context from the process itself, or it can use one of the PVMPI communication routines such as pvmpi_send, which will look up and use the correct context if it is available. This arrangement enables processes to communicate with each other without the user having to explicitly pass the context to other processes.

An advantage of having fixed groups with known attributes is that PVM is able to choose the correct encoding scheme when message passing, thereby enhancing performance on homogeneous systems automatically.

7 Conclusions

The PVMPI system is not just a solution to difficulties of static MPI-1 applications. Rather, it is a system that allows more flexible control over MPI applications than is currently indicated by the MPI-2 forum.

More important, it allows the user to construct sections of an application from different MPI implementations that match different hardware systems. Thus, the user is not forced to run the whole application upon a single system with a single implementation.

In its most simplistic mode of operation, only two or three additional calls are required to fully interoperate entirely different systems. Upgrading the PVMPI system to support new MPI implementations requires only simple changes to current tasker and resource management processes.

The intercommunication operations make using the PVMPI system more akin to the spirit of the original MPI system, especially when it uses contexts in PVM3.4.

References

[1] A. L. Beguelin, J. J. Dongarra, A. Geist, R. J. Manchek, and V. S. Sunderam. Heterogeneous Network Computing. *Sixth SIAM Conference on Parallel Processing*, 1993.

[2] Greg Burns, Raja Daoud and James Vaigl. LAM: An Open Cluster Environment for MPI. Technical report, Ohio Supercomputer Center, Columbus, Ohio, 1994.

[3] Henri Casanova, Jack Dongarra and Weicheng Jiang. The Performance of PVM on MPP Systems. Department of Computer Science Technical Report CS-95-301. University of Tennessee at Knoxville, Knoxville, TN. August 1995.

[4] J. Casas, R. Konuru, S. Otto, R. Prouty, and J. Walpole. Adaptive Load Migration Systems for PVM. *Supercomputing'94 Proceedings* , pp. 390-399, IEEE Computer Society Press, 1994.

[5] Fei-Chen Cheng. Unifying the MPI and PVM 3 Systems. Technical report, Department of Computer Science, Mississippi State University, May 1994.

[6] Nathan Doss, William Gropp, Ewing Lusk and Anthony Skjellum. A model implementation of MPI. Technical report MCS-P393-1193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, 1993.

[7] Graham E. Fagg and Jack J. Dongarra. The Restructuring of SP2MPI PVM. Department of Computer Science Technical Report CS-96-323. University of Tennessee at Knoxville, Knoxville, TN. Febuary 1996.

[8] G.E. Fagg, R.J. Loader, P.R. Minchinton and S.A. Williams. Improved Group Services for PVM. Proceeding of *1995 PVM Users Group Meeting*, Pittsburgh, pp.6, May 1995.

[9] Graham E. Fagg, Roger J. Loader and Shirley A. Williams. Compiling for Groups. Proceeding of *EuroPVM 95*, pp. 77-82, Hermes, Paris, 1995.

[10] G. Geist, J. Kohl, R. Manchek, and P. Papadopoulos. New Features of PVM 3.4 and Beyond. Proceeding of *EuroPVM 95*, pp. 1-10, Hermes, Paris, 1995.

[11] IBM AIX Parallel Environment, Parallel Programming Reference, IBM, Kingston, New-York, September, 1993

[12] R. Konuru, J. Casas, S. Otto, R. Prouty and J. Walpole. A User-Level Process Package for PVM. *Scalable High Performance Computing Conference*, pp. 48-55, IEEE Computer Society Press, 1994.

[13] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994. Special issue on MPI.

[14] Jim Pruyne and Miron Livny. "Providing Resource Management Services to Parallel Applications" *Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing*, May 1994.

[15] Jim Pruyne and Miron Livny. "Parallel Processing on Dynamic Resources with CARMI", *Workshop on Job Scheduling Strategies for Parallel Processing*, IPPS 95, April 25, 1995.

[16] W. Rosenberry, D. Kenney, and G. Fisher. *Understanding DCE.* O'Reilly & Associates, Inc., Sebastopol, CA, 1992.

[17] Anthony Skjellum, Nathan E. Doss and Kishore Viswanathan. Inter-communicator extensions to MPI in the MPIX (MPI eXtension) Library. Department of Computer Science Technical Report. Mississippi State University, Mississippi State, pp. 18, August 1994.

[18] Georg Stellner and Jim Pruyne. Resource Management and Checkpointing for PVM Proceeding of *EuroPVM 95*, pp. 130-136, Hermes, Paris, 1995.

[19] Louise Turcotte. "A Survey of Software Environments for Exploiting Networked Computing Resources", *MSSU-EIRS-ERC-93-2*, Enginerring Research Center, Mississippi State University, Febryray 1993.