PVMPi provides Interoperability between MPI Implementations*

Graham E. Fagg  
Department of Computer Science  
The University of Tennessee  
Knoxville, Tennessee 37996-1301

Jack J. Dongarra  
Department of Computer Science  
The University of Tennessee  
Knoxville, Tennessee 37996-1301  
and  
Computer Science and Mathematics Division  
Oak Ridge National Laboratory  
P.O. Box 2008, Bldg. 6012  
Oak Ridge, Tennessee 37831-6367

Al Geist  
Computer Science and Mathematics Division  
Oak Ridge National Laboratory  
P.O. Box 2008, Bldg. 6012  
Oak Ridge, TN 37831-6367

*This submitted manuscript has been authored by a contractor of the U.S. Government under Contract No. DE-AC05-96OR22464. Accordingly, the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.*

*This work was supported in part by the National Science Foundation, Science and Technology Center Cooperative Agreement No. CCR-8809615 and the Mathematical, Information, and Computer Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract No. DE-AC05-96OR22464 with Lockheed Martin Energy Research Corporation.
DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.
PVMP provides Interoperability between MPI Implementations *

Graham E. Fagg †  Jack J. Dongarra ‡  Al Geist §

December 28, 1996

Abstract

Presently, different MPI implementations cannot interoperate with each other. In order to do distributed computing across different vendors' machines now requires that a single MPI implementation, such as MPICH, be used rather than the vendors' own optimized MPI implementations. This talk describes a software package called PVMPI we are developing that allows interoperability of vendors' optimized MPI versions. Our approach builds on the proven and widely ported Parallel Virtual Machine. The use of PVMPI is transparent to MPI applications and allows intercommunication via all the MPI point-to-point calls. PVMPI allows more flexible control over MPI applications than is currently indicated by the MPI-2 forum by providing access to all the process control and resource control functions available in the PVM virtual machine.

1 Introduction

For the past several years, standardization efforts have attempted to address many of the deficiencies of the different message passing systems and introduce a single standard for such systems. These efforts culminated in the first Message Passing Interface (MPI) standard, introduced in June 1994 [14]. Within a year, several different implementations of MPI were available, including both commercial and public systems.

One of MPIs' 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, systems such as PVM [1, 12] were designed primarily for clusters of computers [19], with the primary goals of portability, and ease-of-use. 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 to improve its message-passing performance [4]. Nevertheless, PVM's main strength is its virtual machine model, which provides for transparent heterogeneous distributed computing.

The aim of this work is to interface the flexible process and virtual machine control from the PVM system with the optimized MPI communication systems of several MPI

---

*This work was supported in part by the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615 and the Mathematical, Information and Computer Sciences subprogram of the Office of Energy Research, DOE, under Contract DE-AC05-84OR21400.
†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
§Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831-6367
implementations. Our goals are to provide the capability for MPI applications to span across multiple MPI implementations and hosts in a distributed environment and to provide MPI applications access to advanced process and resource control capabilities.

In this paper we focusing on the flexibility of PVM in terms of process management and heterogeneous communications and how these can be utilized to allow MPI systems to interact. Then we examine how these systems inter-operate, and we address such issues as language binding and transparent implementation of the user interfaces.

2 Virtual Machine Resource and Process Control

The PVM virtual machine is defined to be a dynamic collection of parallel and serial hosts. With the exception of one host in the PVM virtual machine, any number of hosts can join, leave, or fail without affecting the rest of the virtual machine. In addition, the PVM resource control API allows 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.

The PVM virtual machine is very flexible in its process control capabilities. It can start serial, or parallel processes that may or may not be PVM applications, for example, PVM can spawn an MPI application or /bin/date just as easily as it can spawn a PVM application. The PVM process control API allows any process 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 a process is responding; and
- notify an arbitrary process if another disconnects from the PVM system.

In addition to the above virtual machine control functions, PVM provides plug-in interfaces for extending its resource and process control capabilities. This flexibility has encouraged many projects to use PVM in different distributed computing environments [15] such as Mist [13], dedicated schedulers [16], load balancers, process migration tools [5, 18], and dedicated PVM user configurable resource managers such the GRM project [9].

2.1 PVM Group Services

PVM provides the ability for processes within the virtual machine to form into groups identified by a character string name. 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.

To improve performance, PVM enables the groups to be frozen once formed and their details to be cached locally. Fully dynamic group caching has also been tried[10][11].

3 MPI Communicators

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 MPIInit and leave by calling MPIFinalize. Processes calling MPIInit twice may have an undefined behavior. 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 MPI communications can operate only within a communicator.

Once all the expected MPI processes have started a common communicator is created by the system for them called MPICOMM_WORLD. Communications between processes within the same communicator or group are referred to as intra-communicator communications. Communications between different groups are inter-communicator communications. The formation of an inter-communicator requires two separate (non overlapping) groups and a common communicator between the leaders of each group, as shown in Figure 1.

![Diagram of MPI COMM_WORLD](image)

**FIG. 1.** Inter-communicator formed inside a single MPICOMM_WORLD

MPI-1 standard does not provide a way to create an inter-communicator between two separately initiated MPI applications. Each application has its own MPICOMM_WORLD, with no existing communicator bridging the gap between them.

All internal details are hidden from the user. MPI communicators have relevance only within a particular run-time instance. This strategy excludes different MPI implementations from inter-operating.

4 Related Work

Although several MPI implementations are built upon other established message-passing libraries such as Chameleon-based MPICH [7], LAM [3] and Unify [6], none of these
systems allow inter-operation between separate MPI applications across different MPI implementations. LAM 6.0 does allow some interaction between applications using a subset of functions from the dynamic process chapter of the proposed MPI-2 standards document.

The MPIX [17] project from Mississippi State University has some bearing on this research effort in that it extends the capability of current MPI inter-communicators to allow them to be used in collective operations instead of only in point-to-point operations which is an area that will have to be addressed more carefully by the MPI forum in the future.

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 inter-communicators) were not implemented. The final implementation also only allowed a single MPI (SPMD) application to exist, and thus failed to take advantage of PVM as a communications layer that could allow inter-operation.

Unify did, however, 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.

The only project known to the authors that attempts to interconnect MPI applications in a similar way to PVMPI is currently under way at the Computer Centre of the Rechenzentrum Universitaet in Stuttgart[2]. This project attempts to interconnect pairs of MPPs via specialist processes that use standard TCP/IP for communications.

5 A Prototype System
We developed a prototype system to study the issues in interconnecting MPI and PVM. Three separate issues have been addressed:

1. mapping identifiers and managing MPI and PVM IDs
2. transparent MPI message passing
3. start-up facilities and process management

5.1 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}.

Thus, the underlying address mapping utilized is just a translation between these different types of tuple. An initial prototype version of PVMP [8] used such a system without any further translation (or hiding of mixed identifiers).

The association of this tuple pair was achieved by registering the MPI processes into PVM groups by a user level function call. A matching un-associate or leave call was also created.

The functions are made available in both C and Fortran bindings:
info = PVMPI_Register(char *group, MPI_Comm comm, int *handle);
info = PVMPI_Leave(char *group);

call pvmpi_register( group, comm, handle, info )
call pvmpi_leave( group, info )

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 (i.e. take the same named group). 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.

6 PVMPI System

The above prototype could be confusing for a user as it requires the understanding of two different message passing systems, their APIs, semantics and data formats. A better solution is to transparently provide the interoperability to the MPI application using only the MPI API.

As previously stated, MPI uses communicators to identify message universes, and not PVM group names or TIDs. Thus the prototype system could not allow users to utilize the original MPI calls for inter-application communication. The solution to this problem is to allow the creation of virtual communicators that map either onto PVM and hence remote applications or onto real MPI intra-communicators for local communication.

To handle all possible uses of communicators, all MPI routines using these were re-implemented using MPIs profiling interface. This interface allows user library calls to be intercepted on a selective bases so that debugging and profiling tools could be linked into applications without any source code changes.

Making communicators have a dual capability within the individual MPI implementations was impossible without altering the implementations themselves which was very undesirable. Thus the communicators were replaced by PVMPI system allocated communicators that pointed to a hash table of communication parameters. As user were not allowed to manipulate (look inside) these communicators directly, this did not have any impact on their usage of these communicators within applications. Thus their usage was completely transparent as shown in figure 2.

Intra and inter communicator communications within a single applica-
FIG. 2. *MPI profiling interface controlling communicator translation.*

The MPI COMM_WORLD proceed as normal, while inter-application communication proceed by the use of a PVMPI inter-communicator formed by the PVMPI.Intercomm.create function:

```c
info = PVMPI_Intercomm_create (int handle, char *gname, MPI_Comm *intercom);
call pvmpi_intercomm_create (handle, gname, intercom, info)
```

This function being almost identical to the normal MPI inter-communicator create call except that it takes a handle from the register function, instead of a communicator to identify the local group and a registered name for the remote group. The handle is used so that if a receiving group is registered with multiple names they can be differentiated between correctly.

The call is blocking and collective, although a non-blocking version has also been implemented that can time-out or warn if the requested remote group has attempted to start and then failed so that appropriate action can be taken to aid fault tolerance.

PVMPI inter-communicators are freed using the normal MPI functions. They can be formed, destroyed and recreated without restriction. Once formed, they can be used exactly the same as a normal MPI inter-communicator. Except in the present version of PVMPI there is a restriction that they cannot be used in the formation of new communicators. We are studying how to eliminate this restriction in the next release PVMPI.

6.1 Transparent Message Passing

PVMPI inter-communicators allow the full range of point-to-point message passing calls inside MPI. Also supported is a number of data formatting and (un)packing options, including user derived data types (i.e. mixed striding and formats).

The construction of user derived types revolves around PVMP1 making an independent copy of the users data type as it is created. Then, when a user call specifies this in a communications request, this copy external to any MPI implementation is available for use.

This shadowing of MPI internal data structures makes PVMPI far more complex than a simple wrapper around a MPI profiling library. Much attention has been paid to making
this layer as efficient as possible so that it effects (local) intra-communications as little as possible (i.e. a fast table lookup and a function call).

Receive operations however are complex, and inter-communication relies upon adequate buffering at the receivers end, in-line with normal PVM operation.

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[9] 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. 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 MPIlnit, 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 PVM spawn command was not altered, although when interfacing to a resource manager, it is allowed to be called with one of the following additional flags—PVMPILLAM or PVMPI_MPICH—in addition to 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, . . )

7 Conclusions
The PVMPI system is not just a solution to the lack of interoperability of MPI-1 implementations. It is a system that allows more flexible control over MPI applications and more extended capabilities than is currently indicated by the MPI-2 forum.

More important, it allows the user to run sections of an application across different hardware systems and use the vendors’ optimized MPI implementations on each.
In its most simplistic mode of operation, only two or three additional calls are required to fully inter-operate entirely different systems. Upgrading the PVMPi system to support new MPI implementations requires only simple changes to current tasker and resource management processes.

References


