Why Are PVM and MPI So Different?

William Gropp
Ewing Lusk

Mathematics and Computer Science Division
Argonne National Laboratory

Abstract. PVM and MPI are often compared. These comparisons usually start with the unspoken assumption that PVM and MPI represent different solutions to the same problem. In this paper we show that, in fact, the two systems often are solving *different* problems. In cases where the problems do match but the solutions chosen by PVM and MPI are different, we explain the reasons for the differences. Usually such differences can be traced to explicit differences in the goals of the two systems, their origins, or the relationship between their specifications and their implementations.

1 Introduction

Although they came into existence in quite different ways, PVM [4] and MPI [3] are both specifications[1] for libraries that can be used for parallel computing. It is is natural to compare them. Indeed, many useful comparisons have been published [9, 8, 6, 11]. We consider it worthwhile to do so again for two reasons. The most obvious is that some convergence has recently taken place in the functionality offered by the two systems (e.g., dynamic processes in MPI, static groups and message contexts in PVM), and the very different approaches taken in these extensions merit comment. Equally important, however, is the fact that previous analyses have focused on local, feature-by-feature comparisons, describing similarities as well as differences. Such feature-by-feature comparisons can be misleading, particularly when the two systems use the same word for different concepts. For example, an MPI group and a PVM group are really quite different objects, although they have superficial similarities (e.g., in MPI, sources and destinations are relative to a group, while in PVM sources and destinations are always absolute in terms of the "task ids").

Rather than go through each specification on feature-by-feature basis, we will discuss some of the explicit design goals that were established by the MPI Forum before it undertook to specify the details. In many cases these goals dictated details of the specification (such as the contents of individual function parameter lists). Where these details differ from the corresponding details in PVM, the goal-oriented approach can elucidate the sources of the differences. In addition to differences in explicit goals, we will also note a few differences

[1] We treat the Oak Ridge version of PVM as represented by [5, 1] as the PVM specification. MPI is represented by the MPI-2 specification.

more attributable to the origin of the two systems. PVM was the effort of a single research group, allowing it great flexibility in design and also enabling it to respond incrementally to the experiences of a large user community. In addition, the implementation team was the same as the design team, so it was possible for design and implementation to interact quickly. In contrast, MPI was designed by the MPI Forum (a diverse collection of implementors, library writers, and end users) quite independently of any specific implementation, but with the expectation that all of the participating vendors would implement it. This required that all functionality be negotiated among the users and a wide range of implementors, each of whom had a quite different implementation environment in mind.

2 MPI's Goals

The first task of the MPI Forum was to define the goals that would guide its subsequent discussions. Some of these goals (and some of their implications) were the following:

- MPI would be a library for writing application programs, not a distributed operating system. This goal has implications for resource management issues, as discussed in Section 4.
- MPI would not mandate thread-safe implementations, but its specification would allow them. Thread safety implies that there can be no notion of a "current" buffer, message, error code, etc. As the "nodes" in the network become symmetric multiprocessors, thread safety becomes increasingly important in a heterogeneous, networked environment.[2]
- MPI would be capable of delivering high performance on high-performance systems. Hence, no memory copies would be mandated by the design. Scalability, combined with correctness, for collective operations required that groups be "static".
- MPI would be modular, to accelerate the development of portable parallel libraries. Modularity has many implications. For example, all references must be relative to a module, not the entire program. Hence, process source/destination must be specified by rank in a group rather than by an absolute identifier and context must not be a visible value. There are many others, some of which are described below.
- MPI would be extensible to meet future needs and developments. This led to an object-oriented style without a commitment to an object-oriented language. This approach required functions to manipulate the objects, which is one minor reason for the relatively large number of functions in MPI.
- MPI would support heterogeneous computing (the MPI_Datatype object allows implementations to be heterogeneous), although it would not require that all implementations be heterogeneous.

[2] There is a project to join threads with PVM (TPVM [2]), but this is more a lightweight process model than a fully threaded model and, as such, does not offer as rich a programming model as a fully thread-safe model would.

– MPI would require well-defined behavior (no race conditions or avoidable implementation-specific behavior).

Finally, the MPI Forum sought to simplify the interface by making each approach solve as many problems as possible. For example, datatypes solve both heterogeneity and noncontiguous data layouts, both for messages and for files.

3 Implementation and Definition

One common confusion in comparing MPI with PVM comes from comparing the specification of MPI with the implementation of PVM. Standards specifications tend to specify the minimum level of compliance, while any implementation offers more functionality. In the MPI Forum, many such "added-value" features are listed as expected of a "high-quality implementation".

Error handling and recovery are a good example. Standards tend not to mandate specific behavior on errors, other than to list error indicator values. The expectation is that high-quality implementations will give users what they expect. Specific implementations can easily define their individual handling of errors. Thus, most MPI implementations do not simply abort when an error is detected; just as the PVM implementation does, they attempt to provide a useful error indication and allow the user to continue.

Another source of confusion involves features of a particular implementation that are exposed to the programmer. As an example, consider the `pvm_reg_tasker` routine that allows a process to indicate to PVM that it, rather than fork/exec, should be used to start tasks. This is an powerful hook to allow extension of the PVM *implementation* by special applications, such as debugger servers. MPI, as a standard, has no such object, but specific MPI *implementations* can and do provide similar services; for example, the MPICH implementation of MPI provides a process startup hook used by the Totalview [13] debugger. The MPI standard does not specify how implementations are to provide this service; as a standard, it should not. We note that some PVM implementations for massively parallel procesors (MPPs) also do not provide this routine; if the MPI standard had mandated such a routine, any MPI implementation would have to provide it. This is an example of the freedom of PVM to provide features only in some environments; MPI as a standard does not have that freedom.

4 Dynamic Processes

One way to understand the differences between PVM and MPI is to look at the new MPI-2 features for creating and attaching to processes. While the two approaches may seem similar, they are actually quite different. Perhaps the greatest difference is in the handling of resource information that is used to determine where to create the new process. This reflects a difference in the approach to providing distributed operating system support by MPI and PVM. PVM, through its virtual machine (implemented as the PVM demons) provides a simple yet useful distributed operating system. Special interfaces, such as the `pvm_reg_tasker`, allow the PVM system to interface with other resource management systems. MPI does not provide a virtual machine, even in MPI-2. Rather, it provides a way, through a new MPI object (`MPI_Info`), to communicate with whatever mechanism is providing distributed operating system services.

To illustrate the difference, consider the resources that an application may want to specify when creating a new process:

> Any system that can run an RS/6000, AIX 4.y ($y \geq 2$) executable, with 4 memory banks and at least 256 MB of memory, 200 MB of `/tmp`, and a load of < 2, and able to run for 48 hours.

Such a specification is complicated, and probably beyond what would be expected from a parallel programming system. But it is well within the capabilities of advanced resource management systems. How should a parallel computing system interface with such a system? The choices are (a) pick a small subset that all systems can support, (b) define a general and generic, but fully expressive, system, or (c) provide an interface that allows information to be passed, in an implementation-specific manner, to the resource system.

PVM chose (a)[3]; this is the most convenient form for many users, particularly if the default choices are adequate. More demanding users want (b); this gives them the maximum portability without sacrificing too much expressivity. Unfortunately, (b) has two drawbacks—it isn't extensible, and it assumes that there is a well-defined interface that users agree on. This led the MPI Forum, which spent a great deal of time trying to find a solution like (b), to choose (c). In MPI, this is the "info" argument to an `MPI_Comm_spawn` command:

```
MPI_Comm_spawn(worker_program, MPI_ARGV_NULL, universe_size-1,
          info_for_resource_manager, 0, MPI_COMM_SELF,
          &everyone, MPI_ERRCODES_IGNORE);
```

Just like filenames, the specific contents of "info" depend on the implementation. MPI defines a few items, such as working directory and architecture. Other information can be passed directly to the local resource manager. For example, an MPI implementation could provide a way to pass the above example to the resource manager.

Another difference between MPI and PVM shows up in the presence of `pvm_config` and the lack of an MPI equivalent. The `pvm_config` function provides information on the virtual machine. This information can be used by the programmer to attempt to manage resources directly, for example, by specifying particular hosts in `pvm_spawn`. Why doesn't MPI provide a similar function?

The problem is that the information that any command can provide on the environment is immediately out of date. For example, even in PVM, between the

[3] PVM-aware resource managers such as Condor and LoadLeveler can provide these more complex services, but this is outside of the PVM program itself and is specific to the particular resource manager in use.

time pvm_config is called and pvm_spawn is called, another PVM application may have executed pvm_delhosts, thus invalidating the information provided by pvm_config. As the number of items grows larger and more complex, the likelihood that some critical item will be out of date increases (consider space in /tmp or load average).

The MPI Forum discussed this situation at great length, but could find no workable solution. This is an example of a "race condition", a situation in which the user is in a race with other users and the system and where the "expected" behavior depends on the user's winning the race. It is also another example of the tradeoff in user convenience and precise system behavior. It is natural to wish to perform the operations PVM provides. But they cannot guarantee that the resources described will exist when a process is created.

Hence, the MPI_Comm_spawn call combines process creation with information on the needed resources. Combining operations is a classic way to solve race conditions, and this solution is used in many places in MPI. Eliminating race conditions makes many operations in MPI are collective. Note that the PVM 3.4 pvm_newcontext [1] presents a race condition in the delivery of the new context value to other processes; MPI solves this problem by making context creation collective over all processes that will use the context.

Because of the presence of such race conditions, MPI also forms the MPI communicator (roughly similar to a PVM group and context) at the same time as creating the processes. MPI provides an MPI_Comm_spawn_multiple routine that allows MPI to create processes for a large collection of different executables in a single operation, for the same reason.

Another difference is in the use of MPI intercommunicators. An MPI intercommunicator represents two groups of processes that communicate with each other. It is a natural representation for created processes: one group represents the children and one group represents the parents (multiple parents are allowed in MPI to avoid race conditions).

A final difference illustrates how a combination of features can affect future enhancements. PVM 3.4 adds contexts; unlike MPI, these are user-visible integers that may be sent from process to process and otherwise manipulated by the user. They are also guaranteed to be globally unique; PVM can ensure uniqueness because there is a single virtual machine. MPI's contexts are opaque and defined only by their effect in MPI operations; while a simple implementation could make them globally unique, that is not required (and, for scalability reasons, may not be desirable).

Consider the case of two parallel programs that wish to connect to each other. Both MPI-2 and PVM provide a way to do this. But the PVM approach requires that both programs belong to a single PVM virtual machine. The decision to make the PVM context a visible, explicit integer means that programs belonging to different PVMs cannot safely connect (because they may already have the same "unique" context id). It also means that different PVMs cannot be merged into a single PVM, since this again would make previously unique context integers no longer unique. The MPI-2 approach sacrifices some flexibility (explicit, unique context values) for the extensibility offered by a more modular and encapsulated design.

5 Nonblocking Operations

Nonblocking operations (e.g., `MPI_Isend`) are often misunderstood as a "performance" optimization. In fact, these are necessary when constructing any large, complex communication system. These should be distinguished from *asynchronous* operations. A nonblocking operation is simply one that does not block the calling process. An asynchronous one usually implies that the operation continues to take place concurrently with other operations. (Note that the PVM documention sometimes uses "asynchronous" where MPI would use "nonblocking" and sometimes uses nonblocking.) Consider the following program running on two processes:

```
           Process 1                 Process 2
pvm_psend( ..., size, ... ) pvm_psend(..., size, ... )
pvm_precv( )                pvm_precv( )
```

(particularly if `pvm_setopt( PvmRoute, PvmRouteDirect )` has been called). Does this program work? The answer depends on the size of the messages (`size`), the particular platforms (MPP, workstation networks, or symmetric multiprocessors), and even the environment (e.g., free swap space). For short messages, this will almost always work. At some message size, it will fail; the programs will hang, each waiting for the other to execute the `pvm_precv`. This may seem unusual, but programs that process large amounts of data can easily exceed the amount of available buffering.

Again, this is a tradeoff between user convenience and precise behavior by the interface. MPI is careful to specify the kind of buffering behavior and to provide two alternative solutions to the problem of writing reliable programs: a buffered send (`MPI_Bsend`) with a guaranteed amount of (user-controlled) buffering, and nonblocking operations. The degree to which users want such programs to work was shown by the public reaction to the MPI 1 draft that did not provide a buffered send; the MPI Forum added the buffered send to satisfy this need. See [7, 12] for a more detailed introduction to MPI's handling of buffering.

It is worth noting that the Unix socket interface provides a solution much like the MPI nonblocking operations, though somewhat less convenient for the user. A socket can be set so that `read` or `write` returns rather than blocking, using the error code `EAGAIN` (or `EWOULDBLOCK`) to indicate that the operation would block. This allows careful users to avoid deadlock in their applications.

6 Beyond Message Passing

The evolution of parallel computing has taken us beyond simple message passing. One area that MPI-2 has developed is remote-memory operations. These operations support put, get, and accumulate operations in an "one-sided" manner. Maintaining MPI's commitment to heterogeneity, even these analogues of "store into array" are defined to operate in a heterogeneous environment. MPI makes use of MPI Datatypes and a new MPI object, a "window" (`MPI_Win`), to provide this capability. Maintaining MPI's commitment to performance and scalability as well as adaptability to a wide range of environments, MPI-2 introduces a number of ways to synchronize access to the shared data areas, including support for the bulk synchronous programming (BSP) model. PVM provides no similar functionality.

Parallel I/O is another area where MPI-2 provides a rich set of performance-oriented operations. As with all MPI operations, these support heterogeneous systems and allow the user to choose between forms optimized for a particular system ("native") or for interoperation with other environments and MPI implemenations ("external32"). These facilities are fully integrated with MPI's other functions. In PVM's case, while there are some projects like PIOUS [10], there is no integrated parallel I/O capability. This reflects the differences in the orientation of the two systems: many of the parallel I/O functions are collective and are best defined in terms of static groups, such as MPI defines. PVM only recently added static groups, and they are not as fully developed as the groups in MPI.

7 Conclusion

In this short note, we have focused on a few of the many differences between MPI and PVM. We have shown that the differences between MPI and PVM remain profound, despite some convergence. These differences are accountable for if one bears in mind their quite different origins and goals.

Acknowledgements

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

References

1. J. J. Dongarra, G. A. Geist, R. J. Manchek, and P. M. Papadopoulos. Adding context and static groups into PVM. http://www.epm.ornl.gov/pvm/context.ps, July 1995.
2. A. J. Ferrari and V. S. Sunderam. TPVM: Distributed concurrent computing with lightweight processes. In IEEE, editor, *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society Press, 1995. IEEE catalog no. 95TB8075.
3. Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.

4. Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, MA, 1994.
5. Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM 3 Users Guide and Reference Manual.* Oak Ridge National Laboratory, Oak Ridge, TN, May 1994.
6. G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. PVM and MPI: A comparison of features. *Calculateurs Paralleles*, 8(2), 1996.
7. William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* MIT Press, Cambridge, MA, 1994.
8. J. C. Hardwick. Porting a vector library: a comparison of MPI, Paris, CMMD and PVM. In IEEE, editor, *Proceedings of the 1994 Scalable Parallel Libraries Conference*, pages 68–77, IEEE Computer Society Press, 1995.
9. R. Hempel. The status of the MPI message-passing standard and its relation to PVM. *Lecture Notes in Computer Science*, 1156:14–21, 1996.
10. Steven A. Moyer and V. S. Sunderam. PIOUS: A scalable parallel I/O system for distributed computing environments. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 71–78, 1994.
11. William Saphir. Devil's advocate: Reasons not to use PVM. PVM User Group Meeting, May 1994.
12. Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference.* MIT Press, Cambridge, MA, 1995.
13. Web page: Introduction to the totalview debugger. http://www.dolphinics.com/tw/tv/totalview.html.