Goals Guiding Design: PVM and MPI

William Gropp
gropp@mcs.anl.gov

Ewing Lusk
lusk@mcs.anl.gov

Mathematics and Computer Science Division
Argonne National Laboratory

Abstract

PVM and MPI, two systems for programming clusters, are often compared. The 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. For example, we show that the requirement for portability and performance across many platforms caused MPI to choose approaches different from those made by PVM, which is able to exploit the similarities of network-connected systems.*

1 Introduction

The emergence of the cluster as a viable parallel computing platform, even scaled into the supercomputer range, has been enabled by the simultaneous emergence of message-passing libraries that have made it possible to map parallel algorithms onto them in a portable way. PVM and MPI have been the most successful of such libraries.

PVM [10] and MPI [19, 20] are both specifications[1] for message-passing libraries that can be used for writing portable parallel programs. Recent books on building clusters, both for Linux [26] and for Windows [27], contain chapters on using both MPI and PVM. Since there are freely available versions of each, users have a choice, and beginning users in particular can be confused by their superficial similarities. Several comparisons of PVM and MPI have been carried out since the mid-1990s [18, 17, 12, 23, 16]. 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”).

We prefer to analyze the differences in PVM and MPI by looking first at sources of these differences. The structure of this paper is as follows. In Section 2 we review the explicit design goals of the MPI Forum. In Section 3, we review the similarities between PVM and MPI, leading us in Section 4 to discuss the consequences of separating implementation from design. In Sections 5, 6, 7, and 8, we show how these sources have influenced differences between PVM and MPI in the areas of dynamic processes, contexts, nonblocking operations, and portability, respectively. In Section 9 we focus on those aspects of MPI that go beyond the message-passing model. This paper expands an earlier version [16]. Among the additions here are discussions of parallel I/O, the safety of contexts, and a subtle performance issue in multiparty communications.

2 MPI's Goals

Rather than go through each specification feature by feature, 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, out goal-oriented ap-

[1] We treat the Oak Ridge version of PVM as represented by [5, 11] as the PVM specification. MPI is represented by the MPI-2 specification.

proach will elucidate the sources of the differences. In addition to differences in explicit goals, we will note a few differences 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. Moreover, the implementation team was the same as the design team, so design and implementation could 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. Hence, all functionality had to be negotiated among the users and a wide range of implementors, each of whom had a quite different implementation environment in mind.

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 5.

- 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, and so on. As the "nodes" in the network become symmetric multiprocessors, thread safety becomes increasingly important in a heterogeneous, networked environment.[2] Recent experiences from vendor implementations of a thread-safe MPI (in particular, the IBM implementation [30]) confirm that the MPI *design* is thread-safe.

- 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". An open research problem is finding semantic definitions and appropriate algorithms that allow dynamic groups to meet these same requirements.

- 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. Consider a module that solves a system of linear equations on an arbitrary subset of processes; the ability to restrict the module to a subset of processes is needed by domain decomposition methods and for multidisciplinary applications. 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 (see Section 6). Some other implications of modularity are described below.

- MPI would be extensible to meet future needs and developments. This requirement led to an object-oriented approach without a commitment to an object-oriented language. This approach required functions to manipulate the objects, and was one minor reason for the relatively large number of functions in MPI (large here is relative to C and Fortran programs; C++ and Java programmers are used to large numbers of functions).

- MPI would support heterogeneous computing (the MPI_Datatype object allows implementations to be heterogeneous), although it would not require that all implementations be heterogeneous.

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

For simplicity, the MPI Forum sought to make each approach solve as many of these goals as possible. For example, datatypes solve both heterogeneity and noncontiguous data layouts, both for messages and for files. Similarly, communicators combine both process groups with communications contexts.

The MPI standard has been widely implemented and is used nearly everywhere, attesting to the extent to which these goals were achieved. See [15] for a discussion of the importance of these goals to the success of MPI (or any method for parallel programming).

PVM had, with the exception of support for heterogeneous computing and a different approach to extensibility, different goals. In particular, PVM was aimed at providing a portable, heterogeneous environment for using clusters of machines using socket communications over TCP/IP as a parallel computer. Because of PVM's focus on socket-based communication between loosely-coupled systems, PVM places a greater emphasis on providing a distributed computing environment and on handling communication failures.

3 What is Not Different?

Despite their differences, PVM and MPI certainly have features in common. In this section, we review some of the similarities and, in the process, correct some common misconceptions about the MPI specification. In most cases

[2] There is a project to join threads with PVM (TPVM [9]), 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.

these misconceptions arise because of confusion between specification and implementation.

Both PVM and MPI are *portable*; the specification of each is machine independent, and implementations are available for a wide variety of machines, particularly those likely to appear in clusters.

Once a system is portable, the issue of *homogeneity* can be addressed. Can two processes on different machine architectures communicate with one another despite differences in byte ordering in memory or even word length? To this end PVM provides the `pvm_pack/unpack` functions and the datatype arguments to `pvm_send/recv`; MPI does the same with its more general `MPI_Datatype` argument to many routines. Of course, some *implementations* of MPI, particularly those from hardware vendors, may not be used in a heterogeneous environment, but the MPI specification is designed to encourage heterogeneous implementations, and both the MPICH [13] and LAM [2] implementations support heterogeneous environements.

Both MPI and PVM permit different processes of a parallel program to execute different executable binary files. (This would be required in a heterogeneous implementation, in any case.) That is, both PVM and MPI support MIMD programs as well as SPMD programs, although again some implementations may not do so, and launching MIMD programs may be less convenient than launching SPMD programs. Both MPICH and LAM support MIMD programming.

A final issue is that of *interoperability*. This term refers to the possibility of communicating among processes linked with two completely different implementations. We discuss this issue, and provide further comments on portability and heterogeneity, in Section 8.

In summary, both MPI and PVM are systems designed to provide users with libraries for writing portable, heterogeneous, MIMD programs. In comparing issues, one must not confuse the MPI specification with a particular implementation subcase, such as the `ch_p4` device of MPICH, which is widely used on clusters but does not define MPI.

4 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. Specifically, in any system, there are recoverable and nonrecoverable errors. An example of a recoverable error is an illegal argument to a routine, such as a null-pointer or an out-of-range value. A nonrecoverable error is one where the program may not be able to continue. In many applications, accessing an invalid address or attempting to execute an invalid or privileged instruction is nonrecoverable. The MPI standard does not specify which errors are recoverable, though there has been some discussion in this direction. This is an example of the determination of the MPI Forum to maintain maximum portability—mandating any specific behavior would limit the portability of MPI. Note that even for PVM, some systems provide a less "recoverable" environment than others. For example, systems with proprietary interconnects may kill all processes when any one exits.

Another source of confusion involves features of a particular implementation that are exposed to the programmer. 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 and batch schedulers. 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 [29] debugger. The MPI standard does not specify how implementations are to provide this service; as a standard, it should not. At the same time, the experience with TotalView has defined an interface that MPI implementations (not just MPICH) can use, allowing any debugger to access this information [4]. We note that some PVM implementations for massively parallel processors (MPPs) also do not provide the `pvm_reg_tasker` routine. This is an example of the freedom of PVM to provide features only in some environments. As a standard, MPI does not have that freedom. If the MPI standard had mandated such a routine, any MPI implementation would have to provide it. Instead, MPI's explicit goals mandated that it choose portability over certain kinds of functionality.

When we compare implementations rather than an implementation of PVM with the MPI standard, the gap in this type of functionality narrows. For example, MPICH [13], rather than MPI, does provide a way for debuggers like TotalView to access to internal MPICH state on the message queues. Many users want this information, but it raises an interesting issue: How does one define a standard for the internal state of an implementation? For any implementation this can be done, but different implementations may have different internal states. For example, one optimization for communication has the process issuing an `MPI_RECV` send a message to the expected source of the message, allowing the sender to deliver the message directly into the receiver's memory [21]. Should this information be presented to the user? Other implementation choices might eliminate some queues altogether or make it more difficult to find all pending communication operations; in fact, in the MPICH implementation, there is no send queue unless the system has been configured and built to support the message queue service. By not specifying a model of the internals of an MPI implementation, such as defining a "message queue" does, the MPI standard allows MPI implementations to make tradeoffs between the performance and functionality that the users want.

5 Dynamic Processes

One way to understand the differences between PVM and MPI is to look at the MPI 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 PVM and MPI. PVM, through its virtual machine (implemented as the PVM daemons) 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 mandate or define 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. That mechanism may well be a parallel virtual machine; several implementations already use distributed daemons to start and manage MPI jobs. But we emphasize that daemons are not required by the MPI specification. This feature is important for extreme-scale architectures, where the very existence of local daemons may be impractical.

To understand 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 ($y \geq 3$) executable, with 4 memory banks and at least 512 MB of memory, 400 MB of `/tmp`, and a load of < 2, and is able to run for 48 hours, with access to `/home/me` and the runtime libraries for xlf version 3.4.5 or 3.4.6 but not 3.4.7 or 3.4.4.

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.[4] These drawbacks 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 specifies a few predefined 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. MPI implementations are required to ignore unrecognized fields; this strategy encourages users to provide extra information when possible. Note that the `MPI_Info` object is also used in the file I/O section of MPI-2 to provide performance hints. This is another example of MPI using the same feature to solve multiple goals.

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 time `pvm_config`

[3] PVM-aware resource managers such as Condor and LoadLeveler can provide more complex services, but this is outside of the PVM program itself and is specific to the particular resource manager in use. Portable PVM programs cannot rely on such services.

[4] Several systems are specific to particular resource managers such as LoadLeveler and LSF (Load Sharing Facility), but there is no consensus on which of these, or which combination of features, should be adopted.

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). In the PVM case, the impact of this problem is somewhat mitigated by the fact that each user has a personal parallel virtual machine. Of course, a single user may have multiple parallel jobs running at the same time (e.g., under the control of a system to explore a parameter space), so the problem is not eliminated by providing single user virtual machines.

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. Naturally, one would like to perform the operations PVM provides. But one 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 approach for solving race conditions, and this solution is used in many places in MPI. Eliminating race conditions makes many operations in MPI collective. Note that the PVM 3.4 pvm_newcontext [5] 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. Note that the race is *removed* by this approach, not just moved into the MPI implementation.

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. For the same reason, 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.

Another difference in the handling of process creation 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). In PVM, created processes have only one parent; this reflects PVM's use of the fork/exec or system spawn model of process creation as separate from connecting processes for communication.

6 Contexts

Writing parallel programs is notoriously difficult. One solution is to accelerate the development of parallel libraries, with the expectation that end users will access parallelism through libraries rather than by invoking message-passing functions directly. Thus an original goal of the MPI design was to provide the functionality needed by libraries and missing in most message-passing systems of the time.

The single greatest impediment to the use of parallel libraries has been the lack of modularity. In its simplest form, this impediment manifests itself when a message sent by a library is received unexpectedly by either user code or another library. The solution lies in *contexts* [8]. (Readers not familiar with the notion of context should see the discussion of contexts in Section 2.3 of [14].)

The treatment of contexts illustrates how a combination of features can affect future enhancements. Following MPI, 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 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 again previously unique context integers would no longer be unique. Using an external service (such as a context value server) to allocate contexts simply pushes the problem to a different level without solving it. In addition, there is the very real issue that users may choose to ignore the problems of distributing a visible message context and pick a fixed value. This can lead to subtle problems and was one reason that the MPI Forum made the context value opaque. The MPI approach sacrifices some flexibility (explicit, unique context values) for the extensibility offered by a more modular and encapsulated design. The PVM design is backward-compatible but not as safe.

7 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. They should be distinguished from *asynchronous* operations. A nonblocking operation is simply one that does not block the calling process. An asynchronous operation usually implies that it continues to take place concurrently with other operations. (Note that the PVM documentation sometimes uses "asynchronous" where MPI would use "nonblocking" and sometimes uses nonblocking.)

MPI provides an extensive set of nonblocking operations (`MPI_Isend`, `MPI_Irecv`, `MPI_Ibsend`, etc.). PVM does not provide nonblocking operations in the MPI sense (`pvm_nbrecv` is really what MPI would call a "probe"). MPI provides such operations not only to allow for overlapping communication, but also to make it easier to write portable, correct programs.

Consider the program running on two processes shown in Figure 1, in the case where `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, the program will almost always work. At some message size, on the other hand, it will fail, since the messages must be buffered *somewhere* outside the program itself; 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, tradeoff exists between user convenience and precise behavior by the interface. MPI is careful to specify the kind of buffering behavior and to provide alternative solutions to the problem of writing reliable programs: a buffered send (`MPI_Bsend`) with a guaranteed amount of (user-controlled) buffering, or 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 [14] and [25] for a more detailed introduction to MPI's handling of buffering.

The MPI Forum attempted to define the conditions when `MPI_Send` could be safely used (and in fact, most vendors currently document these and provide some control by way of environment variables). Defining such conditions, however, requires mandating a particular implementation model. The most obvious model is not scalable in its use of memory; more complex models are harder for users to work with and further constrain implementations.

We note 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` to indicate that the operation would block. This allows careful users to avoid deadlock in their applications. POSIX also defines a form of nonblocking operation even more like the MPI nonblocking operations: the `aio_read`, `aio_write`, `aio_error`, `aio_return`, and `aio_cancel` interface for asynchronous I/O. These routines have a test operation (`aio_error` returns 0 when an operation is complete and `EINPROGRESS` when not complete) and a cancel operation. Asynchronous I/O has been used for years in large-scale scientific computing; the MPI approach is not unusual.

A more subtle need for nonblocking operations comes from considering the performance of communication patterns involving more than two processes. Consider four processes communicating with the program

```
MPI_Irecv( ..., nbr1,..., &request[0] );
MPI_Irecv( ..., nbr2,..., &request[1] );
MPI_Send( ..., nbr3, ... );  /* 1 */
MPI_Send( ..., nbr4, ... );  /* 2 */
MPI_Waitall( 2, requests, statuses );
```

This code looks fine but has a subtle problem. If the sends labeled with the comment `/* 1 */` on two processes target the same receiver, then they may suffer a performance degradation because of limits on how fast any process can receive data (for example, limited by network bandwidth). If instead the code was

```
MPI_Irecv( ..., nbr1,..., &request[0] );
MPI_Irecv( ..., nbr2,..., &request[1] );
MPI_Isend( ..., nbr3, ..., &request[2] );
MPI_Isend( ..., nbr4, ..., &request[3] );
MPI_Waitall( 4, requests, statuses );
```

the MPI implementation can send the data for the sends using `request[2]` and `request[3]` at the same time, maximizing the use of the available network bandwidth. Accomplishing the same efficient use of the network resources is possible with blocking operations but requires very careful ordering of operations (and hence much more difficult programming) than in the nonblocking case.

8 Portability, Heterogeneity, and Interoperability

Portability refers to the ability of the same source code to be compiled and run on different parallel machines. *Heterogeneity* refers to portability to "virtual parallel machines" made up of networks of machines that are physically quite different. *Interoperability* refers to the ability of different implementations of the same specification to exchange messages. In this section we compare PVM and MPI with respect to these three properties.

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

Figure 1. Head-to-head communication

Both PVM and MPI had portability as an original goal. As we have seen, MPI's very strict adherence to this principle prevented it from having some features desirable on workstation networks precisely because they could not be implemented in all environments. PVM, defined primarily by a single implementation for workstation networks, has more freedom to add features appropriate for that environment, but at the cost of making some PVM programs not portable to more restrictive environments.

Portability is an underappreciated issue. PVM is considered by many to be highly portable, and in fact the PVM group has done an excellent job in providing implementations across a wide range of platforms, covering most Unix systems and Windows [24]. But the designers of MPI had to consider running on systems that were neither; in fact, MPI has even been used in embedded systems (see `http://www.mc.com`). MPI could not assume that any particular operating system support was available; the design of MPI reflects this constraint. Some users have complained that MPI does not mandate support for certain Unix features, when in fact features such as standard input, process creation, and signals are absent in many important, non-Unix systems.

Support for heterogeneity is provided in both specifications. PVM has separate functions to pack specific data types into buffers; MPI uses basic and derived datatypes. The MPI specification does not mandate heterogeneous support, however; that is up to the implementation. LAM [2], CHimP [1], and MPICH [13] are implementations of MPI that can run on heterogeneous networks of workstations.

Interoperability is outside the scope of the user program, and entirely up to the implementation. Some vendor implementations of PVM are neither heterogeneous nor interoperable with the Oak Ridge version of PVM. The MPI standard does not mandate implementation details, and thus MPI implementations, of which there are many, typically are not interoperable.

Thus, "interoperability" of MPI matches that of PVM. Versions of the *same* implementation (Oak Ridge PVM, MPICH, or LAM) are interoperable. True interoperability is among completely *different* implementations, matched at the level of the wire protocol.

A separate effort (not part of the MPI Forum) has developed an "interoperability standard" called IMPI that provides sufficient standardization for some implementations details so that implementations conforming to this standard can exchange messages. IMPI is now available [3] and several vendor implementations exist.

9 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 a "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 uses 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. These functions have already been implemented by several vendors (HP, Fujitsu, and Cray). 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 implementations ("external32"). These facilities are fully integrated with MPI's other functions. In PVM's case, while there are some projects such as PIOUS [22], no integrated parallel I/O capability exists. This situation 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 eventually added static groups, but they are not as fully developed as the groups in MPI, which has a comprehensive set of operations for manipulating and performing collective communication and computation using scalable algorithms. MPI datatypes have also proved to be critical in obtaining high performance in I/O operations [28].

PVM provides more support for fault tolerance and recovery by exposing to the programmer some of the properties of sockets. MPI does less, in the interest of greater portability. Fault tolerance in MPI is an important research topic. The work on FT-MPI [6, 7] has shown what can be done if one is willing to change some of the fundamental semantics of the MPI specification.

10 Conclusion

In this paper 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.

Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38. The authors also thank the referees for their valuable comments.

References

[1] R. Alasdair, A. Bruce, J. G. Mills, and A. G. Smith. CHIMP/MPI user guide. Technical Report EPCC-KTP-CHIMP-V2-USER 1.2, Edinburgh Parallel Computing Centre, June 1994.

[2] G. Burns, R. Daoud, and J. Vaigl. LAM: An open cluster environment for MPI. In J. W. Ross, editor, *Proceedings of Supercomputing Symposium '94*, pages 379–386. University of Toronto, 1994.

[3] I. S. Committee. IMPI - interoperable message-passing interface, 1998. `http://impi.nist.gov/IMPI/`.

[4] J. Cownie and W. Gropp. A standard interface for debugger access to message queue information in MPI. In J. Dongarra, E. Luque, and T. Margalef, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1697 of *Lecture Notes in Computer Science*, pages 51–58. Springer Verlag, 1999.

[5] 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.

[6] G. Fagg and J. dongarra. Fault-tolerant MPI: Supporting dynamic applications in a dynamic world. In J. Dongarra, P. Kacsuk, and N. Podhorszki, editors, *Recent Advances in Parallel Virutal Machine and Message Passing Interface*, number 1908 in Springer Lecture Notes in Computer Science, pages 346–353, 2000. 7th European PVM/MPI Users' Group Meeting.

[7] G. E. Fagg, A. Bukovsky, and J. J. Dongarra. HARNESS and fault tolerant MPI. *Parallel Computing*, 27(11):1479–1495, Oct. 2001.

[8] R. D. Falgout, A. Skjellum, S. G. Smith, and C. H. Still. The *multicomputer toolbox* approach to concurrent BLAS and LACS. In J. Saltz, editor, *Proceedings of the Scalable High Performance Computing Conference (SHPCC)*, pages 121–128. IEEE Press, April 1992. Also available as LLNL Technical Report UCRL-JC-109775.

[9] A. J. Ferrari and V. S. Sunderam. TPVM: Distributed concurrent computing with lightweight processes. In *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing, August 2–4, 1995, Washington, DC, USA*, pages 211–218. IEEE Computer Society Press, 1995.

[10] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Network Parallel Computing*. MIT Press, Cambridge, Mass., 1994.

[11] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3 Users Guide and Reference manual*. Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, May 1994.

[12] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. PVM and MPI: A comparison of features. *Calculateurs Paralleles*, 8(2), 1996.

[13] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI Message-Passing Interface standard. *Parallel Computing*, 22(6):789–828, 1996.

[14] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface,* 2nd edition. MIT Press, Cambridge, MA, 1999.

[15] W. D. Gropp. Learning from the success of MPI. In B. Monien, V. K. Prasanna, and S. Vajapeyam, editors, *High Performance Computing – HiPC 2001*, number 2228 in Lecture Notes in Computer Science, pages 81–92. Springer, Dec. 2001. 8th International Conference.

[16] W. D. Gropp and E. Lusk. Why are PVM and MPI so different? In M. Bubak, J. Dongarra, and J. Waśniewski, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1332 of *Lecture Notes in Computer Science*, pages 3–10. Springer Verlag, 1997. 4th European PVM/MPI Users' Group Meeting, Cracow, Poland, November 1997.

[17] 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: October 12–14, 1994, Mississippi State University, Mississippi*, pages 68–77, Silver Spring, Maryland, 1995. IEEE Computer Society Press.

[18] R. Hempel. The status of the MPI message-passing standard and its relation to PVM. In A. Bode, J. Dongarra, T. Ludwig, and V. Sunderam, editors, *Parallel Virtual Machine, EuroPVM '96: Third European PVM Conference, Munich, Germany, October 7–9, 1996: proceedings*, volume 1156 of *Lecture Notes in Computer Science*, pages 14–21. Springer-Verlag, 1996.

[19] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *International Journal of Supercomputer Applications*, 8(3/4):165–414, 1994.

[20] Message Passing Interface Forum. MPI2: A Message Passing Interface standard. *International Journal of High Performance Computing Applications*, 12(1–2):1–299, 1998.

[21] K. Morimoto, T. Matsumoto, and K. Hiraki. Implementing MPI with the memory-based communication facilities on the

SSS-CORE operating system. In V. Alexandrov and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1497 of *Lecture Notes in Computer Science*, pages 223–230. Springer, 1998.

[22] S. 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.

[23] W. Saphir. Devil's advocate: Reasons not to use PVM, May 1994. PVM User Group Meeting.

[24] S. L. Scott, M. Fischer, and A. Geist. PVM on windows and NT clusters. In V. Alexandrov and J. Dongarra, editors, *Recent advances in Parallel Virtual Machine and Message Passing Interface*, volume 1497 of *Lecture Notes in Computer Science*, pages 231–238. Springer, 1998.

[25] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI—The Complete Reference: Volume 1, The MPI Core,* 2nd edition. MIT Press, Cambridge, MA, 1998.

[26] T. Sterling, editor. *Beowulf Cluster Computing with Linux*. MIT Press, 2002.

[27] T. Sterling, editor. *Beowulf Cluster Computing with Windows*. MIT Press, 2002.

[28] R. Thakur, E. Lusk, and W. Gropp. A case for using MPI's derived datatypes to improve I/O performance. In *Proceedings of SC98: High Performance Networking and Computing*, Nov. 1998.

[29] Web page: Introduction to the TotalView debugger. http://www.dolphinics.com/tw/tv/totalview.html.

[30] D. Treumann. Personal communication, 1998.