MPI-2: Extending the Message-Passing Interface

Al Geist, ORNL
William Gropp, MCS Division, ANL
Steve Huss-Lederman, ANL and the U. of Wis.
Andrew Lumsdaine, U. of Notre Dame

Ewing Lusk, MCS Div., ANL
William Saphir, NAS
Tony Skjellum, Mississippi State U.
Marc Snir, IBM Corp.

Abstract

This paper describes current activities of the MPI-2 Forum. The MPI-2 Forum is a group of parallel computer vendors, library writers, and application specialists working together to define a set of extensions to MPI (Message Passing Interface). MPI was defined by the same process and now has many implementations, both vendor-proprietary and publicly available, for a wide variety of parallel computing environments.

In this paper we present the salient aspects of the evolving MPI-2 document as it now stands. We discuss proposed extensions and enhancements to MPI in the areas of dynamic process management, one-sided operations, collective operations, new language binding, real-time computing, external interfaces, and miscellaneous topics.

1 Introduction

During 1993 and 1994, a group of parallel computer vendors, library writers, and application scientists met regularly to define a standard interface for message-passing libraries. The result of this effort was MPI (Message-Passing Interface) [8]. Implementations of MPI are now widely available, including portable and freely available implementations [2, 4, 9] and specialized versions from vendors. General information on MPI is available at [1]. For the purposes of this paper, it will be useful to refer to the result of the initial MPI standardization effort as "MPI-1."

MPI-1 defined an interface for a specific *message-passing model* of parallel computation, in which a fixed number of processes with disjoint address spaces communicate through a co-operative mechanism (when two processes communicate, one sends and the other receives). MPI provides many types of point-to-point communication, to incorporate requirements for robustness, expressivity, and performance. Messages are strictly typed and scoped, allowing for communication in a heterogeneous environment. MPI also contains an extensive set of collective operations, process topology functions, and a profiling interface.

The most distinctive feature of the current MPI-2 proposals described in this paper is that they go beyond the strict message-passing model defined above. In MPI-2, processes may create other processes, so that the number of processes in an MPI computation can change dynamically (Section 2). Processes can interact directly with the memory of other processes (Section 3). Extensions, semantic modifications, and subset definitions in support of real-time and embedded systems (Section 4) also represent changes to the computational model.

Other topics being discussed in MPI-2 include extending MPI-1's collective operations to intercommunicators and nonblocking operations (Section 5), bindings for C++ and Fortran 90 (Section 6), and interface definitions for some of MPI's opaque objects so that they can be used more effectively in support of profiling and other libraries (Section 7). Finally, a number of issues, such as interlanguage communication, a portable startup mechanism, and minor repairs to the MPI-1 specification (Section 8), are under consideration in MPI-2.

In the rest of this paper, we present an overview of each of these areas. We assume familiarity with the current MPI Standard. In the Conclusion we describe the current status of these proposals and prospects for their early appearance in implementations.

2 Dynamic Process Management

MPI-1 describes how a group of processes can communicate with one another. It does not specify how those processes are created, nor does it allow processes to enter or leave a parallel application after the application has started. This static process model enables the specification of deterministic semantics and facilitates efficient implementations of MPI.

Nevertheless, a number of important applications cannot use MPI-1 because of the constraints imposed by its static process model. These include manager-worker applications, where the number and type of workers are not known until the manager has started, task farms, applications that can adapt to changing resources, applications with varying resource requirements, and client/server applications. Much of the impetus for relaxing the static process model comes from the PVM community, which is familiar with PVM's relatively rich support for dynamism.

2.1 History

The challenge for MPI-2 was to extend the MPI-1 model subject to the following constraints:

- MPI-1 applications should continue to run unchanged.
- MPI-2 should not compromise the performance or determinism of MPI-1.
- MPI-2 functionality should be applicable to a very wide range of systems, from heterogeneous networks of workstations to massively parallel processors.
- MPI-2 should not take over operating system responsibilities.

The last item turned out to be the most difficult one to deal with. From the beginning, MPI-2 recognized as external to MPI the *resource manager*, which essentially decides where a process is allowed to run, and the *process manager*, which starts and manages processes. The idea was that MPI would not perform the functions of the resource or process manager, but would interact with them through a simple interface.

An early draft of the MPI-2 dynamic process chapter contained new opaque objects, `MPI_Resource` and `MPI_Process`, which played a key role in the interface to resource and process managers. The resource management discussion foundered on the difficulty of devising an interface that allowed meaningful interaction with the wide variety of resource management systems. These range from traditional batch systems on MPPs, to load sharing systems, to parallel operating systems, to a user using `rsh`.

In the end, the Forum decided to remove resource objects and all the routines for managing resources within MPI. Resource management can take place through direct interaction with the resource manager, rather than through MPI routines. Process objects were also removed, and replaced by the MPI-1 (group, rank) process representation. Much of the process management interface remains in MPI, largely to facilitate the management of non-MPI processes in task farms.

A continuing challenge for MPI-2 is to explain why MPI *still* does not provide the functionality of PVM–in particular, the ability to manage a virtual machine. The answer is that PVM is really two products in one–a message passing library and a parallel operating system. MPI addresses message passing, but consciously does not address operating system issues. An intriguing (and plausible) scenario is a portable MPI-2 application, with MPI's efficient and robust message passing, running on a network of workstations managed by PVM. The MPI application could run almost unchanged in another environment where PVM is not fully supported or appropriate.

2.2 The Interface

As currently proposed, the MPI-2 dynamic process interface allows MPI applications to start new processes (including non-MPI processes), send them signals, and find out when they die or become unreachable. The interface also provides a mechanism to establish communication between two independently running MPI applications.

A fundamental concept in MPI-1 is `MPI_COMM_WORLD`, which defines the communication space containing all processes in an MPI application. With MPI-2's ability to add more processes to an application, the definition is modified to be the communication space containing all processes started together. Groups of newly started processes each have their own unique `MPI_COMM_WORLD`, but they also have an intercommunicator that allows them to merge with their parent group, forming a single bigger communicator. MPI-2 also provides an attribute, `MPI_UNIVERSE_SIZE`, that suggests how many new processes might usefully be spawned in the environment.

Separate methods are proposed for starting MPI processes and non-MPI processes. `MPI_SPAWN` starts MPI processes and establishes communication with them, returning an intercommunicator. `MPI_SPAWN_INDEPENDENT` starts processes (which may or may not be MPI processes) but does not establish communication with them. It returns a group. These are blocking routines that start multiple copies of a single binary. MPI-2 includes several variants of these routines. Some versions, for example, start several different binaries (or the same binary with different arguments) under the same `MPI_COMM_WORLD`. All these routines have nonblocking variants to allow processes to do useful work while processes are being spawned.

In order to manage processes and to provide simple fault tolerance, an MPI application must be able to be notified when a process exits and to send signals to running processes. Two new functions do not require communicators and thus can be used with non-MPI processes as well. The function `MPI_SIGNAL` is expected to be able to deliver the full range of POSIX signals to an arbitrary group, but MPI-2 mandates only that the KILL signal be supported. The function `MPI_NOTIFY` provides a general method for an MPI process to be notified when a process exits or becomes unreachable. The notification can be instigated by a hardware failure, software error, or normal exit on completion.

A powerful new functionality being added to MPI-2 is the ability to establish contact between two groups of processes that initially do not share a communicator and may have been started independently. This functionality would be useful, for example, in enabling a visualization tool to start up and attach to a running simulation, or in enabling two parts of a large application, started separately at two different sites to communicate with each other. The collective functions `MPI_CONNECT` and `MPI_IACCEPT` create an intercommunicator that allows the two groups to communicate.

3 Remote Memory Access

The message-passing communication paradigm requires explicit involvement of two processes (sender and receiver), in order to transfer data from the memory of one to the memory of another. Remote Memory Access (RMA) extends the communication mechanisms of MPI by allowing the transfer to occur with the explicit involvement of only one of the two processes.

3.1 Motivation

Remote memory access facilitates the coding of some applications with irregular communication patterns. One situation occurs when a distributed-memory application needs some randomly accessed read-only shared memory (for large shared tables). Some of the processes can be used as "memory servers", while the other processes access the data by using get calls. Another situation occurs with a distributed-memory code where the data distribution is fixed or slowly changing, but where the pattern of use changes dynamically. Each process can compute what data it needs from remote processes and generate the required receives. To generate the matching sends, one needs to compute the inverse of the receive mapping, a time-consuming process that requires all processes to coordinate the data exchange. The use of get calls avoids the need for sends. A generic example is the execution of an assignment of the form `A = B(map)`, where `map` is a permutation vector, and `A`, `B`, and `map` are distributed in the same manner.

RMA can be supported on distributed memory systems by an "RMA agent" at the target node that accepts RMA requests and performs the required read or write accesses in the memory of the target process. A portable implementation might use an asynchronous receive handler to implement this RMA agent. Systems with dedicated put/get hardware (for example, the Cray T3D) could take advantage of that hardware, at least for simple transfers. Systems with communication coprocessors can take advantage of that coprocessor in order to run the RMA agent without interfering with the application processor at the target node.

On shared-memory systems, if the caller can directly access the memory of the target process, RMA can be implemented without an RMA agent: the caller process can directly copy data to or from the memory of the target process.

3.2 Interface Summary

The current MPI-2 draft proposes the following RMA operations:

Put: transfer data from caller memory to target memory

Get: transfer data from target memory to caller memory

Accumulate: update variables in target memory by values from the caller memory. The update operation is an associative operation such as addition or minimum.

Read-Modify-Write: update variables in target memory by values from the caller memory, and return the initial value of the target memory variables. With a suitable choice of the update operation, one obtains synchronization operations such as test-and-set, fetch-and-add, or compare-and-swap.

In addition, a generic asynchronous handler mechanism is provided. This mechanism can be used for a software implementation of remote memory access, as well as for implementing many other communication paradigms. However, the very generality of this mechanism prevents many implementation optimizations that are possible for the more specific RMA operations.

3.3 Issues

Many design choices must considered in choosing a specific syntax and semantics for RMA operations. We list below some of the issues being debated in the MPI-2 forum.

Allocation of RMA windows: On shared-memory systems, it may be efficient to restrict RMA to dynamically allocated, shared-memory segments. On distributed-memory systems, it is more natural to open RMA windows into memory areas that have been already allocated. Both approaches are supported in the current proposal.

Scatter/gather: On systems where communication is relatively expensive, it is important to provide a scatter/gather capability, so that one communication can access data in noncontiguous locations at either ends. The current proposal uses MPI derived datatypes for that purpose. The disadvantage is that more complex scatter/gather patterns may need to be implemented in software.

Synchronization: A message-passing communication achieves both data movement and synchronization of sender and receiver. In contrast, pure shared-memory communication separates data movement from synchronization. Which approach should we follow for RMA? The former choice makes more sense for systems where communication is relatively expensive. The current proposal allows a synchronization counter at the target process memory to be incremented, as part of the RMA operation.

Heterogeneous systems: Like all of MPI, the RMA functions should be supported on heterogeneous systems. This requires that a process be able to describe data layouts (for scatter/gather) in another's process memory, even if that other process runs on a different processor architecture. To achieve this goal, we restrict each RMA window to consist of an homogeneous array of single type entries.

4 Real-Time Extensions to MPI

MPI has helped to promote performance-portable programming of traditional high-performance computing and cluster systems. It has also proven desirable to leverage the success of MPI on parallel applications in the real-time community.

Taking advantage of this opportunity, a number of new organizations and the existing MPI Forum participants initiated an effort to explore what "real-time MPI" might look like. It is not expected that real-time MPI will be a required part of the MPI-2 Standard or that all HPC and cluster MPI implementations will support the real-time profiles.

4.1 Current Discussions

Three specific profiles have been identified as targets for further exploration: *time-based*, *priority-based*, and *event-based* real-time MPI. Most work so far has focused on the first two profiles. Event-based real-time MPI, which is perceived to be a superset of the time-based profile, is beyond the scope of the current discussions.

4.2 Current Issues

For both profiles, the API's are under development. In this section, we briefly outline what has been discussed and decided so far.

Time-Based Profile For the time-based profile, it has been tacitly accepted that an outside calendar must be provided, in addition to the MPI services, in order to schedule the computations associated with this profile of MPI/RT. The calendar will specify when to start MPI communication. The anticipated strategy is to extend the MPI interface by using persistent communications that support this timed startup of communication. Timeout-based communication also will be supported in this way.

Priority-Based Profile Priority-based messaging and threading are commonly occurring strategies in real-time and non-real-time systems. Priority levels are supported by various operating systems and by certain message-passing networks, though not widely. Furthermore, some network systems support virtual channels, which themselves may provide a mechanism of reservation, if not priority, for given "flows" of data.

For this profile, the types of priority are mapped into two categories: per communicator and per send-receive pair. Both categories have advantages. *Priorities based on communicators* (the MPI-relevant view) seem foreign to real-time programmers; yet there is a potential for providing priority inheritance between communicators in a systematic way currently not done in practice. The less controversial approach, *pairwise message priority*, is closer to existing practice.

The subcommittee is trying to draw analogies from other existing efforts, including POSIX efforts, for real-time strategies. Many other messaging approaches for real-time systems are being reviewed as well.

Furthermore, a minimal process model will have to be specified in which processes themselves have priority. In the current MPI system, one can imagine that all processes have equal priority. Introducing a minimal process model may cause deadlocks and performance anomalies such as priority inversion effects, but is essential to making a meaningful profile.

Combined Profile Support From the perspective of a time-division multiplexing of parallel hardware, and scheduling of a backplane for communication, it is conceivable to envision priorities as well. Hence, one may eventually consider having both time- and priority-based support in concert. This is beyond what the committee currently is considering.

4.3 Other Possible Outcomes

The growth of multimedia systems, including simultaneous video and sound, have led to extensions of Internet protocols. The RTP (Realtime Protocol) [11] is a good example. Developing RTP analogs for MPI-2 or beyond is a likely task for the real-time subcommittee, but such features may well have wider appeal and move into the main part of MPI-2 or later efforts. For instance, the RTP approach to message passing supports lossy protocols. This feature is useful in systems where dropping an occasional video frame, for instance, is acceptable.

5 Collective Communication Extensions

MPI-1 has a rich set of collective operations, but they are subject to a number of restrictions. MPI-2 is considering generalizing them is a number of directions.

5.1 Asynchronous Operations

In the current draft, each collective operation specified by MPI-2 has an asynchronous analog. A wide variety of MPI-2 features use asynchronous collective operations on both intracommunicators and intercommunicators.

At present, only one particularly thorny issue remains. Given an intracommunicator (and its underlying process group), should it be possible to have part of the group use the asynchronous collective operation, whereas the rest of the group uses the blocking form? The argument advanced in favor of this mode of operation is ease of use by programmers; the argument against it is that this mode of operation could compromise the ability of blocking collective operations to be optimized in certain respects. The analog of this issue, for intercommunicators, has also not been resolved.

5.2 In-place Operations

A number of collective operations that use both input and output buffers could be respecified or extended to support overwriting of buffers. This issue is sometimes coupled with aliasing issues of specific programming languages, which sometimes disallow it (as in Fortran 77). Nonetheless, there is a significant interest and value in considering these operations, which have been specified in the intracommunicator form and also mentioned in the context of intercommunicators.

The purpose of in-place operations is to allow the user of the collective communication to specify the needed memory for the operation without having to obtain additional buffer storage.

5.3 Intercommunicator Collective Operations

The purpose of intercommunicator collective operations is to support broadcast, reductions, and other operations, extended to include the two-group model of parallel processing offered in MPI-1 by intercommunicators.

Original proposals for extending intercommunicators to support collective operations, in addition to their MPI-1 point-to-point facilities, were first based on [12], which included model implementations.

The additional functionality came in three forms: more collective constructors and manipulators, what is now called "half-duplex" intercommunicator operations that extend intracommmunicator collective operations, and virtual topology-oriented versions of both the constructors and the communication procedures.

The half-duplex operations have been augmented recently by a full-duplex strategy. The full-duplex strategy is to subsume the half-duplex approach once details concerning the API and these operations have been worked out. Asynchronous versions of these operations, as discussed above for intracommunicators, will also be specified.

5.4 Other Possible Outcomes

Since the intercommunicator operations can be considered in the domain of dataflow programming (e.g., for signal processing), where they can be used to advantage, issues of underlying protocols, buffering, and synchronization have been raised. These issues tie in closely with the programming requirements such systems. One potential outcome is that a parallel, stream-oriented protocol be considered in addition to the message-oriented protocol of MPI. This is consistent with [11] but most probably beyond the scope of MPI-2. It indicates the need for continued research and study, and possible standardization at a later date.

Another area of consideration for MPI-2 is the addition of persistent collective operations. Persistent collective operations, like their point-to-point counterparts, would support reuse of setup information, allowing some implementations to create a "planned collective transfer" paradigm.

6 Language Bindings

MPI-1 specifies a procedural interface for both C and Fortran 77. The C interface is robust, but the Fortran 77 interface necessarily violates some Fortran 77 rules, due to limitations in the language. Both languages are being superseded by their modern supersets C++ and Fortran 90. MPI-2 provides language bindings that allow an MPI program to be written in either of these languages. Both C++ and Fortran 90 are extremely powerful languages: each provides one with ample opportunity for error. The designs for the bindings for these languages were therefore guided by the requirement that there be a simple and clear one-to-one correspondence between the language-independent specification of MPI functions and the language-specific bindings of those functions.

6.1 C++ Bindings

The C++ language is an object-oriented extension to the C programming language and has become the most popular object-oriented language in use today [3, 13]. Since C++ is a superset of C, one approach to providing C++ bindings is to simply reuse the C bindings. However, this approach discards much of the expressive power of C++. On the other hand, the C++ interface to MPI could take the form of an actual class library. While such an approach might be attractive for C++ programmers, a class library is too high-level to be considered as an actual set of bindings. The approach taken by MPI-2 for C++ bindings falls between these two extremes.

The design of MPI itself is very much object-based, and the C++ bindings are based on the underlying object-based design principles. The bindings define a small set of classes corresponding to the fundamental object types in MPI with the functionality of MPI provided as member functions of these objects. This interface is fairly lightweight and seeks to meet the requirements of a language binding while still using advanced features of the target language. For instance, MPI error codes are still returned by function calls, no new types of objects are introduced, and the type arguments to function calls must be explicitly provided. Thus, only minimal use of advanced features of C++ such as polymorphism would be available to MPI programmers. This is an approach similar to that taken in [7]. A full-fledged class library that uses such advanced features has been developed in conjunction with the bindings and can be found at [10].

6.2 Fortran 90 Interface

Fortran 90 adds a wide range of features to Fortran 77. These include the module facility, derived types, array syntax, dynamic memory allocation, "pointers", the ability to do strict type checking, and function overloading. At first glance, it seems that MPI-2 should be able to make wide use of these new features. Unfortunately, most of them are too "high level" for MPI to use, and many in fact cause more problems than they solve.

The MPI-2 approach to Fortran 90 bindings therefore focuses more on trying to avoid introducing new problems than on trying to solve old ones. Perhaps the most important addition is a requirement for an "MPI" module for the Fortran 90 interface. Since Fortran 90 is a superset of Fortran 77, it takes the Fortran 77 bindings and data structures as a starting point. A few problems introduced by Fortran 90 are quite difficult to solve. For instance, when a Fortran 90 array section is passed to a "Fortran 77" function (one with no interface block), it is generally copied in and out. This procedure makes it impossible for MPI's nonblocking calls to work, because they rely on knowing the address of the original data—something that is unknowable inside the Fortran 77 routine. The MPI Forum is currently exploring ways to address this particular problem, possibly with a set of overloaded F90 functions for each intrinsic type.

7 External Interfaces

MPI-1 has a number of features that allow users to layer various capabilities on top of MPI. For example, user-defined reduction operations allow the programmer to use MPI for all communication requirements but still perform specialized reduction operations.

7.1 Generalized Requests

MPI-1 had nonblocking operations for basic point-to-point send and receive calls. MPI-2 is proposing nonblocking calls for all collective operations, many one-sided operations, and dynamic spawning. Although these significantly expand the areas covered by nonblocking operations, users still may want additional nonblocking operations. For example, in the current MPI-IO effort [5, 6], nonblocking read and write operations are proposed. It would be advantageous to offer a standard MPI mechanism to perform these additional nonblocking operations. This would allow the use of other MPI features such as MPI_WAIT, reducing the effort in creating such requests and allowing one to control both types of nonblocking operations together.

To this end, MPI-2 has adopted a generalized request mechanism. It allows users to create new nonblocking operations inside of MPI. Such a request is created and freed in an analogous way to how cached information is placed on communicators in MPI-1. At the time of request creation, the user supplies functions that are called when the request is initialized, started, completed, and freed. These callback functions specify what actions need to be taken for the specified nonblocking operation. For example, suppose one wishes to create a specialized nonblocking permutation using the MPI-2 put mechanism. When it is started, it will need to begin nonblocking put operations. This task can be done with the function that is specified to be called when the operation is started.

As with persistent requests, generalized requests are begun with the current MPI_START functions and completed with MPI_{WAIT|TEST} functions. Actions to be taken when parts of the nonblocking operation complete are performed by the communication handlers proposed in MPI-2. The generalized request completes when the user calls MPI_REQUEST_MARK_COMPLETE during one of the communication handler calls.

7.2 Access to Opaque Objects

From the beginning, the MPI Forum has encouraged development of tools that are layered on top of MPI. For example, in MPI-1, the profiling interface was designed to allow profilers to be easily layered on top of MPI. The success of MPI-1 has led to the development of several profiling tools. Many other tools and libraries are also being layered on top of MPI.

One area that has caused difficulties in writing portable tools is the information stored with opaque objects. MPI-1 was deliberately designed with opaque objects. These allow flexibility in implementations and allow for future enhancements without changing the user's view of objects already present in MPI. To allow users to gain access to needed information in opaque objects, MPI has a number of accessor functions. For example, MPI_GET_COUNT will return the number of entries received as stored in the opaque part of the status object. One drawback to this approach is that only information with explicit accessor functions can be obtained in an easy and portable way from an MPI implementation. In MPI-1, the MPI Forum included all the accessor functions that seemed to be needed by users. However, tool writers have noted that they need access to information not typically needed by users. For example, a profiling library often needs the length of a message begun by MPI_START for a persistent request.

To enable these tools to be truly portable, MPI-2 includes a number of functions to expose information stored in opaque objects. These functions are as follows:

- Determination of request type (MPI_REQUEST_CLASS). This call returns the type of request represented by the object. This is useful, for example, to find out what type of request completes in a call to MPI_WAIT.

- Communicator ID (MPI_COMMUNICATOR_ID). This call returns an integer ID for the communicator given. Since implementations do not currently have to keep a unique ID for each communicator, the function MPI_COMMUNICATOR_ID_UNIQUE can be used to find out if the ID given is unique. If MPI required a unique ID, an MPI implementation might be slower in creating communicators. Thus, a balance was struck between the needs of tool writers and the performance of MPI.

- Items associated with a send or receive request. Profilers need information associated with requests to be able to efficiently log the information pertaining to a persistent request when it is started. This same information can be obtained easily from the send or receive call for non-persistent requests. The information available in MPI-2 is the tag (MPI_REQUEST_TAG), the partner (destination in a send and source in a receive) (MPI_REQUEST_PARTNER), and the message length (MPI_REQUEST_LENGTH).

- Composition of a derived datatype. Many tools and users wish to be able to decode a derived datatype in order to determine what are its components. A string representation can be gotten from (`MPI_GET_CHAR_DATA_TYPE`) or put into (`MPI_DATA_TYPE_FROM_CHAR`) an MPI datatype.
- True extent of a datatype (`MPI_TRUE_EXTENT`). Since the extent of a derived datatype can be manipulated via `MPI_UB` and `MPI_LB`, it can be difficult to know the amount of memory required to store a given derived datatype. This call returns the "true" size of a datatype so that this information is available.

Finally, the external interface definition in MPI-2 allows a generalization of the MPI-1 caching mechanism to allow caching on additional handles. The same calls are used but in MPI-2 apply to `MPI_COMM`, `MPI_DATATYPE`, and `MPI_GROUP`.

8 Miscellaneous

A number of topics are being considered by the MPI-2 Forum that do not fall into the categories above.

8.1 Interlanguage communication

In MPI-1, although both C and Fortran-77 bindings were defined, nothing was specified regarding the interoperability of these two languages. Interoperability comprises at least three subareas: initialization, passing of MPI opaque objects from one language to another, and sending a message from one language and having it received in the other.

Only one form of `MPI_INIT` need be called. After the call, the MPI library will be completely initialized for all supported languages.

In order to deal with the portability of MPI opaque objects, such as datatypes, communicators, and requests, conversion functions will be provided that convert the language-dependent "handles" to 32-bit integers and back again. These integers will be portable (among languages) versions of the objects they reference.

Sending a message from a Fortran program to a C program or vice versa will be explicitly allowed, as long as the signatures of the datatypes match. Here we are aided by the fact that the elementary datatypes defined in MPI-1 are distinct in the two languages, and no equivalence (such as one that might exist between the C datatype `int` and the Fortran datatype `INTEGER` on some machines) is assumed. Thus, in sending messages between programs written in different languages, one sends data of a given MPI datatype; no automatic conversion takes place.

8.2 Other Topics

New Datatypes The MPI-1 Standard specifies that when there are "holes" in an MPI derived datatype, the holes may not be overwritten when the message is received. This is appropriate behavior when the holes represent gaps in a strided datatype, for instance, but not when they represent the padding that arises in structures. Performance could be increased if the implementation is allowed to consider such datatypes contiguous. In order to convey to the MPI implementation that the holes are not significant, and can be both send and received, the new datatype `MPI_Type_contiguous_struct` has been introduced.

mpirun Most systems require some special command in order to start parallel programs, and a number of MPI implementations already use `mpirun`. It is proposed that there be a portable mechanism for starting MPI programs. For example,

```
    mpirun -np 64 a.out
```

should be at least one way to start an MPI program `a.out` with an `MPI_COMM_WORLD` of 64 processes on any MPI system.

Repairs to MPI-1 A number of small changes to MPI-1 are being contemplated, particularly in bindings for character string arguments in Fortran. These changes can be traced to the fact that the Fortran-77 bindings for MPI-1 are at variance with the official Fortran standard.

Version Numbers The MPI Standard is not immutable, though we might wish it so. Already minor differences exist between MPI 1.0 and MPI 1.1. It appears likely that users and especially library writers will need to know both at compile time and at run time the version of the MPI specification that their implementation supports. Constants and functions will be supplied for this purpose.

9 Conclusion

We have described the current state (February, 1996) of MPI-2 discussions. The precise content of MPI-2 remains to be decided in the coming months. Although a few implementors are beginning to experiment with some of the notions described here, most are waiting to see what the final specification will look like. The MPI-2 features will be more difficult to implement than those of MPI-1. Nonetheless, enough discussion has taken place that it is possible to discern the likely scope of the functionality that MPI-2 will add to MPI-1. In this paper we have described that functionality.

References

[1] World Wide Web MPI home page. `http://www.mcs.anl.gov/mpi/standard.html`.

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

[3] Grady Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin Cummings, 1994.

[4] Greg Burns, Raja Daoud, and James Vaigl. LAM: An open cluster environment for MPI. In John W. Ross, editor, *Proceedings of Supercomputing Symposium '94*, pages 379–386. University of Toronto, 1994.

[5] Peter Corbett, Dror Feitelson, Yarsun Hsu, Jean-Pierre Prost, Marc Snir, Sam Fineberg, Bill Nitzberg, Bernard Traversat, and Parkson Wong. MPI-IO: A parallel file I/O interface for MPI, version 0.3. Technical Report NAS-95-002, NAS, January 1995.

[6] Peter Corbett, Yarsun Hsu, Jean-Pierre Prost, Marc Snir, Sam Fineberg, Bill Nitzberg, Bernard Traversat, Parkson Wong, and Dror Feitelson. MPI-IO: A parallel file I/O interface for MPI, version 0.4. `http://lovelace.nas.nasa.gov/MPI-IO`, December 1995.

[7] Nathan E. Doss, Purushotam V. Bangalore, and Anthony Skjellum. MPI++ : Issues and Features. In *Proceedings of OONSKI '94*, January 1994.

[8] The MPI Forum. The MPI message-passing interface standard. `http://www.mcs.anl.gov/mpi/standard.html`, May 1995.

[9] William Gropp and Ewing Lusk. User's guide for `mpich`, a portable implementation of MPI. Technical Report ANL-95/6, Argonne National Laboratory, 1996.

[10] Andrew Lumsdaine, Brian M. McCandless, and Jeffrey M. Squyres. Object-oriented MPI, 1996. `http://www.cse.nd.edu/Ĩsc/research/oompi/`.

[11] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications, January 1996. Internet Engineering Task Force RFC 1889; Network Working Group – Standards Track.

[12] Anthony Skjellum, Nathan E. Doss, and Kishore Viswanathan. Inter-communicator extensions to MPI in the MPIX (MPI eXtension) Library. Technical report, Mississippi State University — Dept. of Computer Science, April 1994. Draft version.

[13] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, second edition, 1991.

This work was supported in part 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.