Extending the Message Passing Interface (MPI)*

Anthony Skjellum, Nathan E. Doss,
Kishore Viswanathan, Aswini Chowdappa, Purushotham V. Bangalore
Computer Science Department & NSF Engineering Research Center
Mississippi State University
Mississippi State, MS 39762

Abstract

MPI is the de facto message passing standard for multicomputers and networks of workstations, established by the MPI Forum, a group of universities, research centers, and national laboratories (from both the United States and Europe), as well as multinational vendors in the area of high performance computing. MPI has been implemented already by several groups. Worldwide acceptance of MPI has been quite rapid.

This paper overviews several areas in which MPI can be extended, discusses the merits of making such extensions, and begins to demonstrate how some of these extensions can be made. In some areas, such as intercommunicator extensions, significant progress has been made by us already. In other areas (such as remote memory access), we are merely proposing extensions to MPI that we have not yet reduced to practice. Furthermore, we point out that other researchers are evidently working in parallel with us on their own extension concepts for MPI.

1 Introduction

The MPI Forum introduced MPI as the message-passing standard for multicomputers and networks of workstations in the Spring of 1994. The forum consisted of representatives from universities and national laboratories (from both Europe and the United States), as well as vendors in the area of high performance computing [12]. At the conclusion of the MPI Forum's most recent meeting (February 1994), a set of topics for further study was informally discussed ("MPI Forum's Journal of Development"). These areas were principally extensions to MPI that could be extrapolated from the standardization process, but which were explicitly omitted from the first version of MPI. For certain features, the omission occurred because of lack of common practice in the area or because the committee thought that their early inclusion would impair MPI's timely completion (*e.g.,* relationship to parallel I/O, process management). Yet for others (*e.g.*, collective operations for intercommunicators), no proposals for such features had been forwarded to the committee by the time MPI was finalized. Finally, actual practical use of MPI over the past year has led to new ideas, ideas which needed time to mature and be discussed outside the standardization forum.

1.1 Overview

MPI encompasses point-to-point and collective message passing, communication scoping (groups and communicators), virtual topologies, datatypes, profiling, and environmental inquiry, within a reliable model of distributed computing [12]. Notably, the following concepts were excluded from the initial MPI standardization effort:

- Active messages,
- Threads,
- Virtual shared memory,
- Process management,
- Parallel input-output,
- Process scheduling,
- Dynamic load balancing,
- Inter-language compatibility,
- Multi-protocol (or multi-vendor) standardization,

*The authors were supported in part by the NSF Engineering Research Center for Computational Field Simulation (NSF ERC), Mississippi State University. This work was also supported in part by DOE Cooperative Agreement DE-FC04-94AL98921.

- Standards for data conversion strategies.

Inclusion of any or all of these issues would arguably have rendered the original MPI standardization effort intractably complex and protracted. Nonetheless, selective extensions to MPI are now seen as definitely beneficial, given MPI's initial popularity and its quickly developing acceptance. Our rationale for extending MPI is therefore as follows:

- A few obvious features were omitted from MPI. features.
- MPI is the notation of libraries, and library writers will benefit from additional specialized features that may be ignored by most application programmers. Such features will, in part, increase performance, and also improve software engineering aspects.
- Better scalability will result by improving certain data structures.
- Better performance portability could result.
- Other models of parallel programming are possible.
- Better connection to forthcoming hardware would result by explicitly supporting remote memory access and active messages at the user level.
- Residual porting issues could be reduced by providing a self-contained computational model, like P4, PVM, etc.

The principal purpose of this paper is therefore to discuss areas of extension for MPI, and, specifically, to describe our own work in some of these areas thus far. This paper overviews areas in which MPI can be extended, discusses the merits of making such extensions, and begins to demonstrate how some of these extensions can be made. Areas of extension that we recognize as important include the following:

- Intercommunicator extensions for collective operations [24],
- Basic extensions of the process model to include spawning [16],
- Thread extensions to MPI (and to the extended intercommunicator calls we propose) [7],
- Interrupt receive calls,
- Remote memory access extensions,
- A user-level, active message interface to MPI.

These topics are actually closely related in some important respects, beyond their superficial relationship as part of experimental and proposed MPI extensions. Thread extensions to MPI lead to a more general model of parallel computing. When combined with intracommunicators and intercommunicators[1], the behavior of this model is easy to explain, and leads to good software engineering characteristics of the parallel software, as well as evident opportunities for enhanced performance (in good implementations). In what follows, we describe the properties of multi-threaded MPI programming, together with some of its impacts on implementations and programming styles.

Furthermore, the availability of MPI's initial version as a "stable intermediate form" [1] has given us a stable base on which to propose these additions, some of which will most probably be proposed with alternative syntax and/or semantics by others in the near future.

We have an especially good opportunity to demonstrate most of these extensions within the Argonne/MSU MPI model implementation (also called MPICH) [9], and others by using the Unify implementation that provides a (currently subset) MPI interface to the PVM 3.x system, while retaining PVM's services [27].

1.2 Latency/Bandwidth Space of Messaging

To put this paper in proper perspective with other efforts in message passing protocols, one wants to consider a two-dimensional design space of latency vs. bandwidth. Given lower limits for latency, and upper limits for bandwidth of different networks, protocols and application programmer interfaces reveal different degrees of performance and functionality to users. We identify some points in the design space, in terms of APIs/protocol suites:

- Extremely low latency/low bandwidth (active messages, remote put/get),
- Low latency/high bandwidth (MPI when fully optimized for hardware, put/get, user-space DMA to hardware, forthcoming "message-way" protocol [22]),
- Medium latency/medium bandwidth (MPICH now on top of portable, but relatively slow drivers, no-copy TCP/IP stacks),
- High latency/medium bandwidth (slightly improved TCP/IP stacks),
- High latency/low bandwidth (standard TCP/IP stack with many copies).

[1] At present, the central importance of intercommunicators in an MPMD programming model is definitely underappreciated.

This above list indicates why TCP/IP is not the ultimate protocol for use with high speed MPI implementations. More subtle is the fact that either active message and remote put/get protocols could be utilized by MPI for high performance, but could also provided to users as portable calls within MPI (perhaps as a combined interface remote access/execution interface[2]).

2 Intercommunicator Extensions

The MPI Forum standardized intracommunicators and intercommunicators. For the former, a single group of processes is involved. For the latter, communication is between two groups. A process in the "local group" always communicates with a process in the "remote group" (see figure 1). The intercommunicator model provides a logical extension of client-server communication, but also has benefits in symmetric protocols between groups of processes. However, the MPI Forum stopped short of providing sufficient capabilities to intercommunicators. Specifically, a small number of constructors were provided, and point-to-point communication was also defined. However, collective communication and topology functions were omitted for intercommunicators, though most of the protocol complexity needed is implied by functions that MPI already encompasses.

Detail of our efforts on single-thread extensions to MPI intercommunicators are given in [24], where we augment functionality of intercommunicators omitted in standard MPI. Here we outline the concepts detailed in [24], and mention these features again in the light of multi-threaded message-passing later in the paper (section 4). Features are provided in several areas:

- extended constructors,
- collective communication operations,
- means to cope with overlapping groups, without multiple threads,
- virtual topology extensions.

2.1 Constructors

MPI provides two communicator constructors, `MPI_Intercomm_create` and `MPI_Comm_dup`, that are applicable to both intracommunicators and intercommunicators. The following constructors are added in

[2]The unified strategy is not what we propose in this paper.

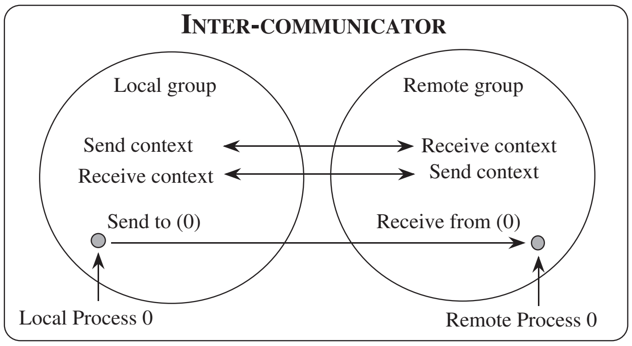

Figure 1: Schematic of an MPI Intercommunicator.

MPIX, extending the intracommunicator constructors to the intercommunicator case:

- `MPIX_Intercomm_create`,
- `MPIX_Comm_dup`,
- `MPIX_Comm_create`,
- `MPIX_Comm_split` (see figure 2).

The first two functions provide additional protocol needed by MPIX to support collective communication on intercommunicators, although each they have an MPI equivalent. The latter two functions have intracommunicator analogies in MPI. Additionally, the following constructors are provided:

- `MPIX_Intercomm_partition`,
- `MPIX_Comm_overlap_create` (see figure 3).

These functions have no MPI analogs at present; the former provides a mechanism to create an intercommunicator from an intracommunicator without resort to a "parent" communication context. The latter call provides a non-deadlocking means to construct an intercommunicator when the local and remote groups overlap[3].

2.2 Collective Communication

In MPI, the point-to-point protocol must be used to communicate between the local and remote groups, or else an intracommunicator must be explicitly constructed in order to use collective communication. This is unnecessarily restrictive. MPIX provides a consistent model for how broadcast, reduce, allreduce, scatter, gather, and allgather should be extended to intercommunicators. Representative operations are

[3]The MPI standard requires the presence of multiple threaded programming to avoid deadlock (if there is group overlap) with the standard call `MPI_Intercomm_create`.

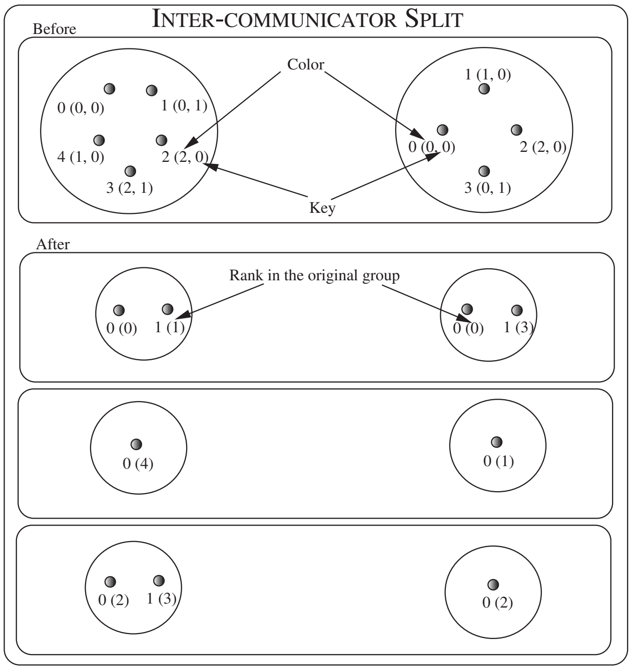

Figure 2: Intercommunicator construction by splitting an existing intercommunicator.

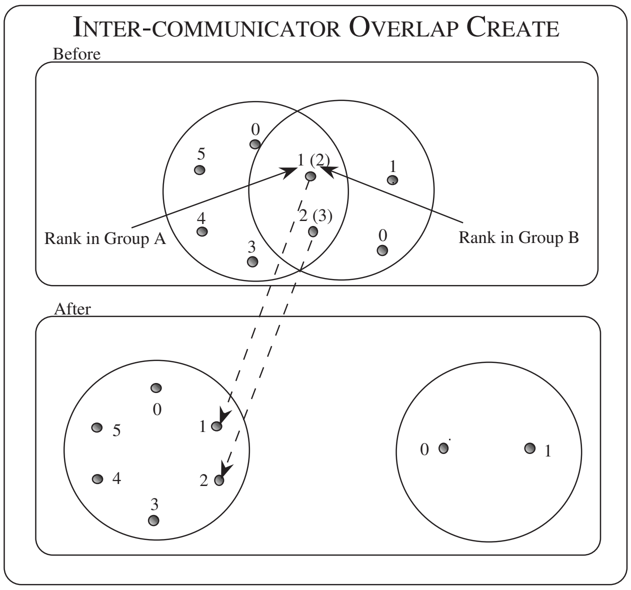

Figure 3: Intercommunicator construction from two overlapping intracommunicators.

shown in figures 4, 5, and 6. Importantly, for operations that have a virtual focus of data between the groups, the intercommunicator paradigm does not require that implementations resort to a single intermediate process. Hence, this paradigm reveals to application-level services the structure of the groups and correlates to the bisection bandwidth available for communication across the group boundaries.

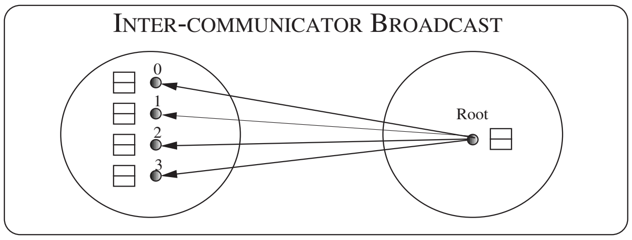

Figure 4: Intercommunicator broadcast.

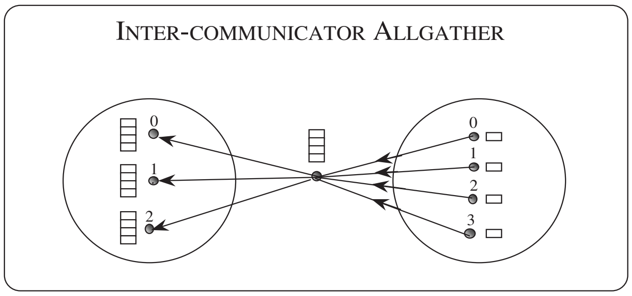

Figure 5: Intercommunicator allgather. The focus of data to one process is representative, not mandated by the semantics.

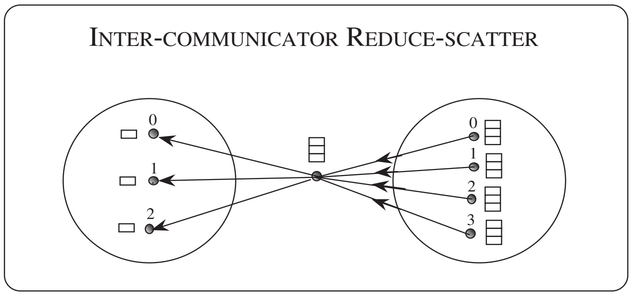

Figure 6: Intercommunicator reduce-scatter. The focus of data to one process is representative, not mandated by the semantics.

2.3 Virtual Topologies

The extension of virtual topologies to intercommunicators completes the MPIX picture, by providing convenient and powerful descriptions of both local and remote groups in terms of cartesian and graph topologies. These extensions, together with topology-relevant constructors (see figure 7), provide an expressive means to cope with multi-disciplinary applications running different programs (such as global ocean, atmosphere models). Figure 8 illustrates partitioning of intercommunicators based on cartesian topologies.

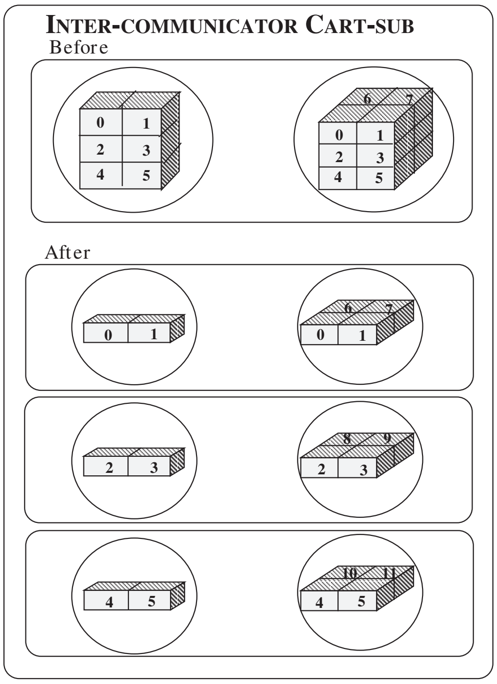

Figure 8: Intercommunicator cartesian partitioning analogous to MPIX_Intercomm_split.

3 Spawning Processes / Worlds

MPI intercommunicators provide structure useful for such strategies as client/server and multidisciplinary applications. They also can serve as the basis for an enhanced dynamic process and host/node process model. Traditionally, in systems such as PVM that have supported a dynamic process model, one process spawns one or more additional processes as needed. In [16], the authors present a model whereby a group of processes collectively spawn processes. Their approach encompasses the traditional approach (the spawning group can be of size one) as well as broadening it (the spawning group can contain multiple processes). We propose a similar function:

```
MPI_Comm_spawn(string,root,comm,&intercomm);
```

The arguments have the following meanings:

string Process specification string that describes "how" and "where" the processes are to be created,

root Process in `comm` that contains a valid `string`. The `string` argument in all other processes is ignored.

comm Original spawning communicator, whose group synchronizes on the spawning of the children processes.

intercomm Resulting intercommunicator. The local group contains the processes found in `comm`; the remote group contains the group of spawned processes.

Figure 9 illustrates how this call might be used. The spawn call is collective over MPI_COMM_WORLD with all processes in MPI_COMM_WORLD receiving an intercommunicator as the result of the spawn call. The MPI_Intercomm_merge call is then used to merge the two separate worlds (represented by the two sides of the intercommunicator) into one intracommunicator (galaxy).

```
main (int argc, char **argv)
{
  int rank, root = 0;
  char *string = (char *)0;
  MPI_Comm intercomm, galaxy;

  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  if (rank == root)
    string = "process specification";
  MPI_Comm_spawn(string,root,MPI_COMM_WORLD,
                 &intercomm);
  MPI_Intercomm_merge(intercomm, &galaxy);
  /* ... */
}
```

Figure 9: Illustration of the use of MPI_Comm_spawn to spawn new processes.

The spawned processes come into existence with an MPI_COMM_WORLD that contains only the newly spawned processes. In other words, the parents and

Existing MPI Functions	Corresponding MPIX Functions
MPI_Cart_create	MPIX_Cart_create, MPIX_Cart_remote_create
MPI_Cart_get	MPIX_Cart_get, MPIX_Cart_remote_get
MPI_Cartdims_get	MPIX_Cartdims_get, MPIX_Cartdims_remote_get
MPI_Cart_rank	MPIX_Cart_rank, MPIX_Cart_remote_rank
MPI_Cart_coords	MPIX_Cart_coords, MPIX_Cart_remote_coords
MPI_Cart_sub	MPIX_Cart_sub, MPIX_Cart_remote_sub
MPI_Cart_shift	MPIX_Cart_shift[a]
MPI_Graph_create	MPIX_Graph_create, MPIX_Graph_remote_create
MPI_Graph_get	MPIX_Graph_get, MPIX_Graph_remote_get
MPI_Graphdims_get	MPIX_Graphdims_get, MPIX_Graphdims_remote_get
MPI_Graph_neighbors_count	MPIX_Graph_neighbors_count, MPIX_Graph_remote_neighbors_count
MPI_Graph_neighbors	MPIX_Graph_neighbors, MPIX_Graph_remote_neighbors

[a]Intracommunicators only.

Figure 7: Existing MPI Topology Functions and their MPIX Counterparts.

children have different MPI_COMM_WORLD's. The following MPI_Comm_parent function allows these newly spawned processes to access the intercommunicator that contains the "parent" group of processe:

```
MPI_Comm_parent(&intercomm);
```

If intercomm is not a valid communicator, then the processes were not spawned with MPI_Comm_spawn. Figure 9 shows example code for the processes being spawned.

```
main (int argc, char **argv)
{
  int rank, root = 0;
  char *string = (char *)0;
  MPI_Comm intercomm, galaxy;

  MPI_Init (&argc, &argv);
  MPI_Comm_parent ( &intercomm );
  if (intercomm != MPI_COMM_NULL)
    MPI_Intercomm_merge(intercomm,&galaxy);
  /* ... */
}
```

Figure 10: Example use of MPI_Comm_parent by spawned processes.

The host/node process model can be considered as a special case where the spawning communicator consists of only one process (*e.g.*, MPI_COMM_SELF). We note that the MPI_Intercomm_merge function used in figures 9 and 10 is not necessary since the resulting intercommunicator can be used for communication. It is shown to demonstrate how to create a new intracommunicator that contains the complete set of existing processes. We summarize this section as follows:

- the parent spawners get an intercommunicator back,
- spawn is a collective operation for spawning processes,
- the children's world is the remote group of the parent's intercommunicator,
- parents and children have a collective relationship immediately,
- this bootstraps to bigger "comm_worlds" with well-characterized semantics, and no race conditions.

4 MPI with Multiple Threads

Currently, MPI supports the case of a single parallel thread of execution, with multiple contexts of communication in that thread. These contexts of communication may have groups that overlap in the thread of execution, and deadlock avoidance is the responsibility of the programmer. However, the great advance of MPI is its ability lexically to scope messages in a well-formed parallel procedure call, and to provide message spaces in a less structured, object-oriented type computation (where persistent objects may reserve safe communication space).

In addition to the single thread of execution, MPI's incomplete (or immediate) send/receive calls (*e.g.*, MPI_Isend and MPI_Irecv) indirectly suggest that there are parallel threads at work in the implementations of such send and receive operations (as required by the progress and fairness specifications of the MPI protocol). Such threads are specific to MPI, because user code cannot execute in such threads; rather, only the pre-specified operations of send and receive are possible "in the background." The predisposition of MPI to support such a background mode of operation, in order to create non-blocking operations and permit overlapping of communication and computation, bears importantly on the generalizations we propose in this work.

4.1 Thread-Oriented Challenges

A number of challenges present themselves when creating thread-oriented message passing.

- One must define a collective thread paradigm that describes progress, fairness, and deadlock.
- One must define logical extensions to MPI that are useful with multiple threads.
- Despite MPI's basic thread-safe interface, one must examine the interactions of threads and MPI.
- One wants to make statements about scheduling properties of the thread package that lead to good performance.

Furthermore, it is clear that threads will be used both for performance reasons (overlapping communication and computation), and for software engineering aspects (convenience of composing parallel libraries in different parallel threads).

We note that maximum performance-oriented threads is given by *hardware latency* $\times$ *Hardware Bandwidth* ("Amdahl's law for threads") [21, 25][4]. This is a measure of the network "concurrency"; simply, this is the maximum number of units of information that can impinge on a process per unit time; if each goes to a single thread, then this is the maximum number of threads one can feed. For each class of latent access (network, I/O, etc), there is separate thread concurrency, motivating the use of multiple threads for different purposes in a complex application. Within the limit of thread concurrency for each kind of latent device, threads are for performance. Beyond limit of thread concurrency, threads are for notation, software engineering of libraries and applications. We need scheduling that reflects the multiple uses of parallel threads for notation and for latency hiding.

[4] with realizable performance further bounded by context switch costs.

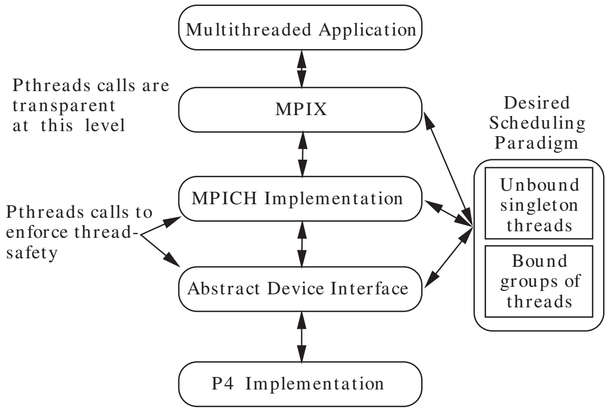

Figure 11: Message passing system hierarchy

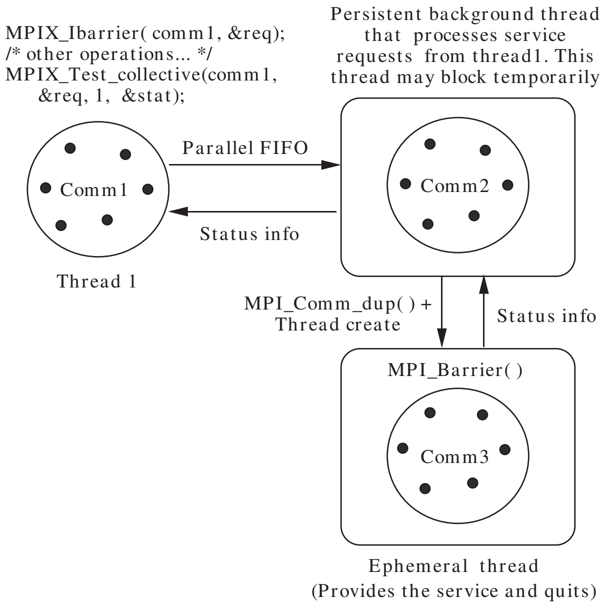

Figure 12: Nonblocking barrier using MPI and multiple, parallel threads (MPIX perspective of thread-safe features of MPI).

4.2 Cases

The following cases arise with the addition of multiple threads per process:

1. one thread per communicator and multiple communicators/threads,
2. multiple threads per communicator and multiple communicators.

Furthermore, there are two classes of threads to be supported:

- short-lived threads, executing extended MPI calls (ephemeral),
- long-lived threads, executing user code.

We describe the properties of each of these models in turn.

4.2.1 Thread Model #1

In the first model, each parallel thread of execution has its own communicator, so that communication never crosses parallel thread boundaries (assuming intracommunicators), and a lexical scoping of messages is retained. This model is readily supported by providing the following capabilities for long-lived threads:

- `MPIX_Comm_dup_thread`, a call that creates a new parallel thread, with a duplicate communication space, and begins executing user code in the new thread. Alternatively, the thread can be made initially stopped, awaiting a later call to `MPIX_Comm_start_thread`.
- `MPIX_Comm_split_thread`, a call that creates a new parallel thread, with a set of disjoint communication spaces, controlled by user-specified `color` and `key`, analogous to `MPI_Comm_split`.
- `MPIX_Comm_free`, a call that destroys the thread associated with the communicator created by `MPIX_Comm_dup_thread`, as well as the communicator, and then returns control to another parallel thread.

This model uses the communicator to keep messages scoped properly. An extra background thread can be cached by application programs and libraries for the purpose of submitting a background collective operation, or for similar capabilities.

Besides this, MPI can be extended to provide the non-blocking collective operations omitted from the initial standard, as depicted for `MPIX_Ibarrier` in figure 12[5] In this figure, MPI is extended to utilize a background thread for any communicator that may pose non-blocking collective operations. The main thread communicates with FIFO command queues through shared memory to the persistent background thread, which accepts asynchronous commands. These commands are handled by using regular blocking MPI functions, as well as thread-oriented operations. Typically, short-lived (ephemeral) parallel threads are dispatched to handle requests, and the permanent background thread may cache duplicate communicators to avoid synchronizations. We note that it is necessary to provide a separate class of test, wait, testany, waitany functions for collective operations, typified in the figure by `MPIX_Test_collective`.

4.2.2 Thread Model #2

In this second model, the full complexity of the first model is supported, plus multiple threads per communicator are permitted. This places a further burden on implementations, particularly collective operations.

MPI requires that it always be safe to issue a collective call, notably a call to get additional safe communication space, `MPI_Comm_dup`. For both this and the first model, the implementation must continue to guarantee when it is safe to use this call. However, it is quite difficult to support collective communication in multiple threads for model #2 because we explicitly have chosen to have multiple threads for a communicator. (In model #1, it is feasible to envisage an encoding of "context" that includes both communication space and thread ID, though other options will also work.)

So, we propose some restrictions on model #2, which we believe will not be restrictive in practice:

- For any communicator, only one collective operation of any kind may be active at any given time or else the program is erroneous.
- Any number of threads may send point-to-point messages in the communicator, and the user may select tag strategies to allow naming of threads or other concepts that relate to the application or library. MPI will not ever try to name threads.

A slightly less restrictive version of this model would only allow one collective operation of each variety per communicator, and this will be plausible for many implementations (particularly those that use a separate context for collective operations, and have tags for such operations within their protocol). Furthermore,

[5] An asynchronous barrier is the simplest possible collective operation, of pedagogical value here, but it is not likely to be used in practice. It is useful to specify other collective operations as "background operations."

reduce and allreduce operations that use different operations could also be allowed to coexist temporally. However, it seems fine, in general, to avoid this added complexity, multiple communicators to achieve multiple, temporally overlapped, collective operations.

4.3 The Role of Intercommunicators

Intercommunicators play an extremely interesting role in an MPI extended with threads. Currently, an intercommunicator create is limited to non-overlapping groups. This restriction is removed for the case of multiple threads executing in the overlap of the two groups. Furthermore, if there is total overlap, intercommunicators provide a means to communicate between parallel threads. Thus, intercommunicators are extremely helpful in coping with inter-parallel-thread communication.

In standard MPI, unlike MPIX, there are no collective communication operations on intercommunicators. Thread-oriented MPIX calls would permit incomplete collective operations between groups, and their regular blocking form would permit broadcasts between overlapping groups of parallel threads. This provides a complete, elegant picture for programming with multiple parallel threads based on overlapping process groups.

5 Interrupt Receive

The basic syntax of an Intel-NX-like `Hrecv()` can be posed easily in an MPI framework:

`MPIX_Hrecv(buf,cnt,type,src,tag,comm,func)`

The semantics of the operation are much more subtle; one has to pose any restrictions placed on `func` needed to keep the entire program thread safe and correct, as well as define the semantics of `func` such that `MPIX_Hrecv` can be implemented on existing architectures and operating systems both correctly and efficiently.

The interrupt receive is a thread-like operation, which implies that `MPIX_Hrecv` will likely require that we provide the user an MPI lock (analogous to `masktrap` provided in NX) to guarantee that critical sections of code don't get interrupted when the hrecv `func` is called (*i.e.*, say the main thread is in a call to `malloc` and `func` also does a malloc). `MPIX_Hrecv`, as illustrated in figure 13, can be treated as an example of an ephemeral thread executing a regular receive operation, a thread that has high priority for execution whenever a message matching it arrives. This reduces this call to a case we have considered previously, except that thread scheduling properties are needed to help performance.

```
MPIX_Hrecv(buf, ..., comm, func)
{
  spawn a new thread;
  if (I'm the parent) return;
  MPI_Recv (buf, ..., comm, &status);
  func (status);
  destroy thread
}
```

Figure 13: Possible Implementation of `MPIX_Hrecv`

The functionality of `MPIX_Hrecv` can be achieved by a thread-safe MPI implementation plus a threads package. If `MPIX_Hrecv` is added to the standard with the semantics we have posed, MPI implementors will be put in a position of having to write a thread-safe, thread-capable implementation. The MPI implementors will then be left with the difficulty of providing a subset of the features found in a thread library as well as making their implementation truly thread safe (especially if the handler function is allowed to do MPI communication operations). We do not want to impose this difficult burden on MPI implementors; therefore, if an `MPIX_Hrecv` is to be accepted into the standard, there must be an accessible, efficient, and correct implementation on wide range of architectures and operating systems. In order for this to be true, we suggest the following criteria for judging proposal for `MPIX_Hrecv` semantics:

- The handler function is executed in the user thread.
- The handler is not guaranteed to execute until another MPI function is called.
- An implementation of `MPIX_Hrecv` doesn't affect the efficiency of other MPI operations.

There is quite a bit of sentiment for an interrupt receive because of its perceived usefulness. We agree that it is a useful and needed operation but are wary of its inclusion in the MPI standard because of implementation issues.

6 Remote Memory Access

Figure 14 illustrates two possible remote memory access operations: a "put" operation that puts a value into another processes' memory and a "get" operation that allows a process to retrieve values from another processes memory. These operations are one-sided operations; only one process of the two involved has to act to perform the operation. MPI currently does not provide these types of operations. At least two processes (or the same process twice – *e.g.*, a process sending a message to itself) must be involved in every communication operation.

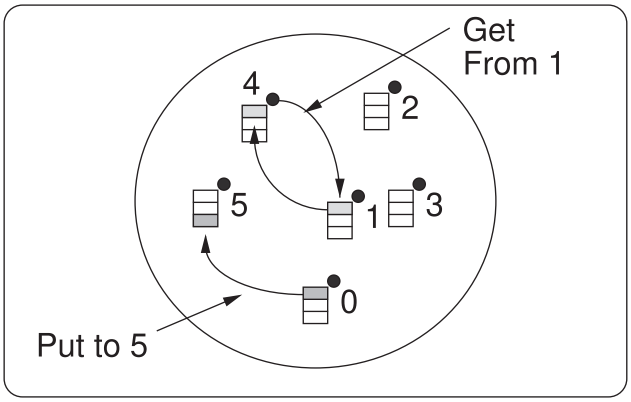

Figure 14: "Get" and "Put" remote memory access operations.

Some machines have lower-level "get" and "put" operations (*e.g.*, Cray T3D) that are likely to be used to implement an efficient version of MPI. Since "get" and "put" operations are examples of interrupt driven message passing and can be implemented given an "hrecv" operation like the one described in section 5, it also possible to implement these operations using threads where lower-level "get" and "put" operations do not exist.

Remote memory access operations have traditionally relied on a shared address space. MPI is not meant to provide a true shared address space, but could allow programmers to register a specific location or "page" in memory that remote access operations will apply to.

```
MPIX_Page_create( start,count,type,comm,
                  &page_comm);
```

The `MPIX_Page_create`[6] function would register a "page" of memory with MPI by providing the `start` location for the page, the `count` of objects in the page (*i.e.*, the length of the page), and the `type` of objects in the page (*i.e.*, an MPI datatype definition). This description would then be associated with a new communicator called `page_comm` that is used for remote access operations. Just as MPI guarantees that collective and point-to-point operations do not interfere with each other on a single communicator, the user is also guaranteed that remote memory access operations will be performed in a separate communication space. Each process in the original communicator can specify a different count. A process is allowed to specify a zero count, thereby publishing no memory for remote access by the rest of the communicator's group.

[6] As with other functions not part of MPI, we consistently use `MPIX_` to distinguish these functions from standard functions.

Once a communicator has been created, the following "put" operations would be permissible:

```
MPIX_Put(buf,count,type,to,offset,
            page_comm);
MPIX_Iput(buf,count,type,to,offset,
            page_comm,&req);
```

The data to be put (written) is described by (`buf, count, type`). The `to` argument indicates the destination for the data, with `offset` describing where in the remote page the data is to be placed. A similar "get" (read) operation can also be described:

```
MPIX_Get(buf,count,type,frm,offset,
        page_comm,&status);
MPIX_Iget(buf,count,type,frm,offset,
         page_comm,&req);
```

The three arguments (`buf,count,type`) describe the final location for the data. The `frm` and `offset` arguments provide the address of the data to be retrieved.

We are not strongly advocating the inclusion of these functions into the MPI standard for some of the same reasons we do not yet propose the inclusion of `MPI_Hrecv` in the standard (see section 5). We do, however, note that it is possible to pose a viable interface for remote memory access features. The benefit and convenience of these operations may outweigh the complications in a future standardization phase.

We note in passing that collective operations on page-oriented communicators might also make sense, subject to the pages of shared memory published by each group member, but we do not expand on this notion at present.

7 User-Level Active-Memory Interface

An active message [28] consists of the following:

- A pointer to a "small" message handler,
- A "small" amount of message data.

They are essentially light-weight remote procedure calls. In a typical active-message model, one generally assumes that:

- Every node has the same program with data and functions in identical memory locations.
- Handlers are atomic operations whose main purpose is to remove messages from the network.
- The arrival of a message generates an interrupt.

As MPI standardizers, we are interested in active messages insofar as they would provide a low-latency, low-bandwidth protocol with no buffering or scheduling, and certain applications might like to build their own higher-level functionality on top of these active messages (such as RPC, remote memory access, coherent shared memory, and so on). Currently, MPI features provide access to medium-latency, high-bandwidth through the send/receive paradigm. Alternatively, extended features, like remote memory access, could also be added directly to MPI, without revealing an interface of the active message type. We have shown this option elsewhere in this paper (see section 6).

We offer some rationale as to why active messages are not in the current MPI version:

- No data conversion for heterogeneous systems is implied in active messages, and automatic data conversion was a requirement of the MPI forum for all protocols.
- Compatible address space assumptions were perceived to be needed.
- The interrupt-driven model was perceived to be needed, which would require standardization of signalling as part of MPI, thereby implicating POSIX issues into MPI.
- The length of active messages could vary dramatically from implementation to implementation, making life difficult for portable programs that just want to use single active messages, rather than build a fragmentation/defragmentation strategy on top of them.

These features caused the committee to steer away from active messages, yet, in retrospect, several of these concerns can be overcome.

We can easily provide a picture of active messages, analogous to that provided by Thinking Machines, on the CM-5. On the CM-5, there is a five, 32-bit word limitation to the active message. The following basic functionality is provided:

```
CMAM_4(node,function,arg1,arg2,arg3,arg4);
CMAM_open_segment(base,count,xfer_fn,xtra);
CMAM_xfer(node,sid,buffer,count);
CMAM_wait(&var,value);
CMAM_poll( );
```

In MPI, function pointers do not make sense across distinct processors, giving rise to the need for function registration as well as segment registration. A possible interface for active messages in MPI might therefore be as follows:

```
MPIX_AM_Function_create(comm,func,type1,
     type2,type3,type4,&func_id);
MPIX_AM_Function_free(&func_id);
MPIX_AM_4(to,func_id,comm,a1,a2,a3,a4);
MPIX_AM_Segment_create(start,count,
     type,xfer_fn,xtra,comm,&seg_id);
MPIX_AM_Segment_free(&seg_id);
MPIX_AM_Xfer(start,cnt,type,dest,comm);
MPIX_AM_Wait(&var,value);
MPIX_AM_Poll();
```

We note that arbitrarily choosing the number of arguments to four in `MPIX_AM_4` serves only as an illustration how the same functionality afforded on the CM-5 might be achieved in an MPI interface. Clearly, different versions of active messages might provide a variety of "natural lengths" for active messages. Yet, providing a variable-length interface might imply additional overhead. Hence, we have chosen a comparable interface to "plain" active messages here.

It is possible and reasonably efficient to remove the requirement of equivalent address spaces across the group, because there is a communicator defining participants, and because there is a required registration process. Specifically, members of a communicator agree that the handle to a function is equivalent across their group, and each places a compatible table entry for this function within the lower level of the AM architecture. Necessarily, `MPIX_AM_Function_create` becomes a synchronization across the group specified within the communicator `comm`.

Other points to note include:

- Data conversions can be accomplished by registering functions with data type information and by opening segments using MPI datatype syntax.
- The CM version of Active Messages does not use interrupts, only polling, so it should be possible to implement MPI versions either with or without interrupts (and not mandate interrupts). A careful look at MPI's requirements of fairness and progress are needed to see if a non-interrupt implementation would comply.
- Addition of this interface to MPI remains consistent with the philosophy of the MPI Forum (*i.e.*, access to high speed protocols such as ready send).

Finally, we note that the provision of active message type functionality in a high level of MPI does not require that active messages be the low-level mechanism by which MPI is implemented.

8 Future Work

Performance and software engineering issues drive both the current and future extensions of MPI. The high-level issues of concern that make an extended MPI more interesting include:

- Temporal composition: Threads allow overlapping of unrelated libraries with explicable, useful program behavior,
- The need for multi-threaded parallel models for effective parallel I/O,
- Intercommunicator collective operations as basis for scalable client-server computing,
- Remote memory access as an extension to the programming paradigm,
- More scalable request structures for algorithms that use many pending messages,
- Process management to integrate MPI into a distributed operating system,
- A standardized queueing interface, to integrate MPI into a distributed operating environment with multiple (conflicting) performance objectives,
- Multi-protocol implementations to allow MPI to run cross-vendor, and across different parts of non-uniform memory access architectures, without relying strictly on TCP/IP and XDR services (low level standards other than TCP/IP would facilitate this).

9 Summary and Conclusions

This paper has covered a number of areas in which MPI, the de facto message-passing interface standard, can and should be extended. We have covered a number of areas, from process management to multiple threads of execution. The combination of extensions to intercommunicators and multiple thread compatibility are an attractive model for parallel I/O, and for scalable client-server computing. The addition of the remote memory access services, and active message services extends MPI into a number of special protocols that will help certain applications, and reveal high-performance to the user within the scoped model of communicator-oriented message-passing.

It is important to note that while some parts of this paper are theoretical, we have actually made significant progress with implementing MPI extensions in the area of intercommunicators. Furthermore, we have made significant practical progress towards incorporating thread-safe communication into the MPICH model implementation of MPI. This work will continue apace.

Acknowledgements

We acknowledge the collaboration of Ewing Lusk and William Gropp on the Argonne/MSU MPI Implementation (MPICH), upon which this much of this research is built. We acknowledge Paula Vaughan, of MSU ERC, for her work on the Unify system, which is also integral to some of the experiments described in the foregoing, or planned for the future.

We acknowledge informal discussions with Jim Cownie of Meiko, Ltd, and Hubertus Franke of IBM T. J. Watson Research Laboratory, relevant to the topics discussed here. We acknowledge further discussions with Rik Littlefield of Pacific Northwest Laboratory (Battelle) concerning interrupt-receive messages.

References

[1] Grady Booch. *Object-Oriented Analysis and Design with Applications*. The Benjamin/Cummings Publishing Company, Inc., 1993.

[2] Joseph Boykin, David Kirschen, Alan Langerman, and Susan LoVerso. *Programming under MACH*. Addison Wesley Publishing Company, 1993.

[3] James Boyle, Ralph Butler, Terrence Disz, Barnette Glickfeld, Ewing Lusk, Ross Overbeek, James Patterson, and Rick Stevens. *Portable Programs for Parallel Processor*. Holt, Rinehart and Winston Inc, 1987.

[4] Ralph Bulter and Ewing Lusk. *A User's Guide to the P4 Parallel Programming System*. Argonne National Laboratory, October 1992.

[5] Greg Burns, Raja Daoud, and James Vaigl. LAM: An open cluster environment for MPI. Available by anonymous ftp from tbag.osc.edu in pub/lam/lam-papers.tar.Z, 1994.

[6] Fei-Chen Cheng, Paula L. Vaughan, Donna Reese, and Anthony Skjellum. The Unify System. Technical report, Mississippi State University — NSF Engineering Research Center, June 1994. Version 0.9.1.

[7] Aswini K. Chowdappa, Anthony Skjellum, and Nathan E. Doss. Thread-safe message passing with P4 and MPI. Technical Report TR–CS–941025, Mississippi State University — Dept. of Computer Science, April 1994. Revised, October, 1994.

[8] Technical Committee on Operating Systems and Application Environments. *Draft Standard for Information Technology - Portable Operating Systems Interface (POSIX)*. IEEE Computer Society, April 1993.

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

[10] Steve Otto et al. *MPI Annotated Reference Manual*. MIT Press, 1994. In preparation.

[11] J. R. Eykholt, S. R. Kleiman, S. Barton, R. Gaulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams. Beyond Multiprocessing... Multithreading the SunOS kernel. In *In proceedings of the USENIX conference*. SunSoft, Inc. Mountain View, California, June 1992.

[12] Message Passing Interface Forum. Document for a Standard Message-Passing Interface. Technical Report Technical Report No. CS-93-214 (revised), University of Tennessee, April 1994. Available on **netlib**.

[13] Hubertus Franke, Peter Hochschild, Pratap Pattnaik, and Marc Snir. An efficient implementation of MPI on IBM-SP1. In *Proceedings of the 1994 International Conference on Parallel Processing*, August 1994.

[14] Hubertus Franke, Peter Hochschild, Pratap Pattnaik, and Marc Snir. An efficient implementation of MPI. In *IFIP WG10.3 Working Conference on Programming Environments for Parallel Distributed Systems*, April 1994. also IBM Research Report: RC 19493(84718) 3/25/94.

[15] William Gropp and Ewing Lusk. An abstract device definition to support the implementation of a high-level point-to-point message-passing interface. (in progress).

[16] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.

[17] T. W. Doeppner Jr. Threads - a system for the support of concurrent programming. Technical report, Brown University, Dept of Computer Science, 1987.

[18] Rik Littlefield, October 1994. Personal Correspondence on Interrupt Receive.

[19] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOS 5.0 Multithread Architecture. Executive Summary, September 1991.

[20] *A Library Implementation of POSIX Threads under UNIX*, January 1993.

[21] Charles L. Seitz, July 1992. Personal Correspondence on Network Concurrency.

[22] Charles L. Seitz, September 1994. Personal Correspondence on Message-Way protocol. 'Message Way' is a trademark of Myricom, Inc.

[23] David Sitsky. Implementation of MPI on the Fujitsu AP1000: Technical Details, September 21 1994. Release 1.1.

[24] Anthony Skjellum, Nathan E. Doss, and Kishore Vishwanathan. Inter-communicator Extensions to MPI in the MPIX (MPI eXtension) Library. Submitted to ICAE Journal special issue on Distributed Computing, July 1994.

[25] Burton Smith, August 1992. Personal Correspondence on Network Concurrency.

[26] A. Tevanian, R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper, and M. W. Young. MACH threads and the UNIX kernel: The battle for control. In *In Proceedings of the USENIX conference*, pages 185–197, 1987.

[27] Paula L. Vaughan, Anthony Skjellum, Donna S. Reese, and Fei-Chen Cheng. Migrating from PVM to MPI, part I.: The Unify System. Technical report, Mississippi State University — NSF Engineering Research Center, July 1994. To appear in the proceedings of Frontiers '95.

[28] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: A mechanism for integrated communication and computation. In *Proc. of the 19th Int'l Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.