Generalized Communicators in the Message Passing Interface

Erik D. Demaine, *Student Member*, *IEEE*, Ian Foster,
Carl Kesselman, *Member*, *IEEE Computer Society*, and Marc Snir, *Fellow*, *IEEE*

Abstract—We propose extensions to the Message Passing Interface (MPI) that generalize the MPI communicator concept to allow multiple communication endpoints per process, dynamic creation of endpoints, and the transfer of endpoints between processes. The generalized communicator construct can be used to express a wide range of interesting communication structures, including collective communication operations involving multiple threads per process, communications between dynamically created threads or processes, and object-oriented applications in which communications are directed to specific objects. Furthermore, this enriched functionality can be provided in a manner that preserves backward compatibility with MPI. We describe the proposed extensions, illustrate their use with examples, and describe a prototype implementation in the popular MPI implementation MPICH.

Index Terms—MPI, process spawning, multithreading, process names.

1 Introduction

An important feature of the Message Passing Interface (MPI) [1], [2] is the communicator, which allows the programmer to define unique communication spaces within which a set of processes can communicate without fear of interference. Communicators are created by collective calls that create a local instance of a communicator object in each of a set of processes. We can think of the local communicator object in each process as a "communication port" that the process can use to send messages to and receive messages from other such "ports" connected by the same communication space.[1] In an intracommunicator, the ports are connected so that each can send to and receive from any other. In an intercommunicator, the ports form two disjointed sets, with each member of one set being able to send to and receive from any member of the other set.

The two related concepts of communication space and communication port are powerful and general. However, we believe that their utility is significantly reduced by the fact that an MPI communicator must define exactly one port per process in a process group and by the fact that only fully connected and bipartite communication structures are supported. Such communication structures are often sufficient for homogeneous, single-threaded, SPMD computations. However, task-parallel, multithreaded, and heterogeneous computations often can benefit from more flexible communication structures. Consider the following situations:

- a multithreaded computation in which a programmer requires unidirectional communication channels between two dynamically created threads of control located in different processes,
- a dynamic computation in which a master process "connects" two dynamically created child processes,
- a task-parallel computation in which communication needs to be directed to a specific data-structure (or object) rather than to a process.

In each of these examples, the collective, all-to-all nature of the MPI communicator is an impediment to a direct expression of the required communication structure.

A more fundamental problem with MPI communicators is their second-class nature. Before a communicator can be created, all member processes must know about each other through other communicators. In other words, a process cannot send the name of another process in an MPI message unless the recipient already knows the name from another communicator of which it is a member. This restriction has serious consequences for dynamic process spawning in MPI-2 [4] as the names of newly created processes cannot be communicated to other processes without "merging" communicators via global synchronization. We argue that, in this situation, *first-class process names* are needed, by which we mean the ability to store a globally significant process name in an MPI message. The following are some examples of situations in which this feature is required:

- a name server that associates process names with service descriptions,

1. This view is related to the notion of "port" formalized in OOMPI [3], an object-oriented package for message passing based on MPI.

- *E.D. Demaine is with the Department of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada. E-mail: eddemaine@uwaterloo.ca*
- *I. Foster is with the Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439 and the University of Chicago, Chicago, IL 60637. E-mail: foster@mcs.anl.gov.*
- *C. Kesselman is with the Information Sciences Institute, University of Southern California, Marina del Rey, CA 90292. E-mail: carl@compbio.caltrech.edu.*
- *M. Snir is with the IBM T.J. Watson Research Center, PO Box 218, Yorktown Heights, NY 10598. E-mail: snir@watson.ibm.com.*

Manuscript received 1 Nov. 1999; revised 20 Nov. 2000; accepted 1 Dec. 2000. For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 110889.

- a group communication system with support for dynamic membership; the group server must be able to notify processes of new members and specify their names in order to facilitate communication,
- concurrent programming languages that support dynamic process spawning and "first-class channels." A channel is a unidirectional connection between a collection of sending processes and a collection of receiving processes, similar to an MPI intercommunicator. First-class channels, that is, channels that can be passed over channels, are an important part of many concurrent languages, including Concurrent ML [5], Fortran M [6], PCN [7], and Strand [8].

In fact, first-class process names are crucial in nearly every situation involving the dynamic creation of processes and/or threads and their lack severely limits the possible applications for MPI.

In this article, we propose a generalized communicator mechanism that eliminates these limitations while maintaining backward compatibility with MPI as currently defined. This generalized mechanism allows a process to create new communication ports and connect these ports in an arbitrary topology. Furthermore, the port becomes a first-class object and can be sent to other processes via MPI messages.

Other extensions to the MPI communicator have been proposed. For example, Skjellum et al. [9] propose mechanisms that allow for a richer set of collective operations over communicators as well as extensions that support multithreaded execution. The extensions presented here are orthogonal to these proposals.

An alternative proposal to communicators are the channels in MPI/RT, a real-time message-passing standard [10]. In MPI/RT, process names are first-class objects that can be communicated in messages, unlike MPI. However, channels themselves (which offer communication contexts) cannot be communicated in messages and have a static and completely connected structure. This paper addresses all of these issues simultaneously.

In the remainder of this article, we introduce our generalized communicator mechanism, illustrate its use with examples, and describe a prototype implementation. A preliminary version of this paper appeared in [11].

2 Generalized Communicators

In MPI, a communicator is first and foremost a global structure. An implementation of this structure typically maintains a set of local data structures, which we might call local communication objects (LCOs). However, no mechanism is provided for manipulating these LCOs directly. Our extensions generalize the MPI communicator so that the LCO becomes an MPI data type in its own right. Thus, the generalized LCO implements the "communication port" abstraction referred to in the introduction. Each LCO contains explicit references to other LCOs and, hence, provides a purely local view of a communication topology.

This new interpretation of the MPI communicator separates the two concepts of communication and process. An arbitrary number of LCOs can be created within a process and communications can be directed to different LCOs within the same process. In addition, the new interpretation makes it possible to create arbitrary communication topologies. These new capabilities enable the use of more general protocols for communication port creation and destruction. For example:

- A multithreaded computation can dynamically define a point-to-point communication namespace between two or more threads of control, whether these threads are located in the same or different processes.
- We can pass references to communication ports ("port capabilities") between processes, thus allowing, for example, a thread to delegate responsibility for performing a particular communication.
- We can define communicator-like structures containing more communication ports than processes. This feature makes it possible to perform collective operations involving multiple threads [12], where the number of threads may be greater than the number of processes, which is a situation that can arise on shared-memory multiprocessors or in programs that create one thread per application "task."

Fig. 1 illustrates some of the communication structures that can be specified using the port construct. We emphasize that the extended interpretation of the local communicator object does not affect MPI's intracommunicator and intercommunicator concepts. For example, an intracommunicator connecting N processes is just a collection of N LCOs, each referencing the N other LCOs.

3 Send and Receive Slots

We now consider the structure of an LCO in some detail. Associated with an LCO is a sequence of *send slots* and a sequence of *receive slots*. A receive slot is a *communication endpoint*, a location to which communication can be directed. A send slot is a reference to a receive slot in an LCO. This reference comprises the LCO's name, which is a new MPI datatype, and the index of the receive slot in the named LCO's receive set. LCOs can be connected to form arbitrary graphs. The only consistency requirement on an LCO is that, for each send slot, there exist an LCO with a matching receive slot.

By interpreting the rank in MPI communication operations as a slot index, rather than the rank of the source or destination process in the process group, we can apply operations such as send and receive to a port without modification. In a send call, the rank specifies the send slot referencing the LCO into which data is to be deposited. In a receive call, the rank specifies the index of the receive slot in which to look for incoming data. If the LCOs are connected in an all-to-all configuration, the behavior is exactly that of a conventional MPI intracommunicator.

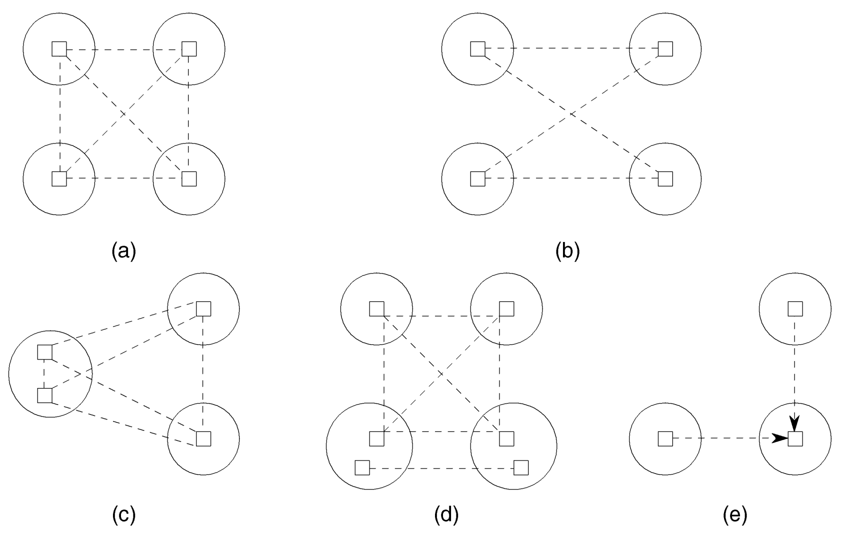


Fig. 1. The two types of commuincation structure can be specified in MPI: (a) the fully connected communicatior and (b) the intercommunicator's bipartite graph. Three different structures that can be specified by using MPI extended to support ports. (c) A fully connected communicator with more than one LCO per process, (d) a regular communicator coexisting with a dynamically created communicator connecting two LCOs, and (e) a communicator structure that allows two senders to communicate with a single receiver.

A local communicator object can be used anywhere that an MPI communicator is used. Hence:

- All MPI point-to-point communication functions can be applied to LCOs.
- All MPI collective communication functions can be applied collectively to a set of LCOs defining an intracommunicator.
- MPI intercommunicator functions can be applied collectively to a set of LCOs defining an intercommunicator.
- MPI functions involving process groups and communicators can be applied to LCOs. This issue is discussed below.

In each of these situations, multiple threads may be required to avoid deadlock if two or more of the LCOs involved in a communication are located in the same process.

The semantics of communication on generalized LCOs are identical to those for MPI communicators. In particular, messages sent on a communication edge linking two LCOs are received in order and communication failure results in an exception at the sending or receiving LCO. In a multithreaded system, if two or more threads perform receive operations that would match an incoming message, all block until the message arrives. The first thread to have performed a matching receive operation then succeeds and receives the message. The others stay blocked.

For notational purposes, we can think of an LCO as a pair with the following form:

$$port = \{set\text{-}of\text{-}send\text{-}slots,\ set\text{-}of\text{-}recv\text{-}slots\}$$

where a set is denoted by a comma-separated list, enclosed in angle brackets, and a send slot has the form

$$send\text{-}slot = lco\text{-}name[recv\text{-}slot\text{-}number]$$

A receive slot is denoted simply by a "+." We use this notation to present some examples.

Example: Channel. A unidirectional channel is defined by a pair of LCOs connected such that one can be used to send to the other. For example, the two LCOs

```
P0 = {<P1[0]>, <>}     P1 = {<>, <+>}
```

define a channel from LCO P0 to LCO P1. P1 has a single receive slot. P0 has a single send slot, which contains a reference to P1's receive slot. Hence, the calls

```
MPI_Send(in, 1, type, 0, tag, P0)
MPI_Recv(out, 1, type, 0, tag, P1, status)
```

will transfer data from `in` to `out`. That is, a send on P0's zeroth send slot is matched by a receive from P1's zeroth receive slot.

Example: Intracommunicator. An MPI intracommunicator is defined by a set of LCOs configured as a fully connected network. For example, the LCOs

```
P0 = {<P0[0],P1[0],P2[0]>, <+,+,+>}
P1 = {<P0[1],P1[1],P2[1]>, <+,+,+>}
P2 = {<P0[2],P1[2],P2[2]>, <+,+,+>}
```

define a fully connected network, that is, an MPI intracommunicator. The calls

```
MPI_Send(in, 1, type, 2, tag, P0)
MPI_Recv(out, 1, type, 0, tag, P2, status)
```

will transfer data from `in` to `out`. That is, a send to P0's second send slot is matched by a receive on P2's zeroth receive slot.

Example: Intercommunicator. An MPI intercommunicator is defined by two sets of LCOs configured such that each LCO in the first set can send to and receive from each LCO in the second set. For example, the LCOs

```
P0 = {<P2[0],P3[0]>, <+,+>}
P1 = {<P2[1],P3[1]>, <+,+>}
P2 = {<P0[0],P1[0]>, <+,+>}
P3 = {<P0[1],P1[1]>, <+,+>}
```

define a structure equivalent to an MPI intercommunicator. In this case, LCOs P0 and P1 are connected to LCOs P2 and P3 so that, for example, the calls

```
MPI_Send(in, 1, type, 1, tag, P0)
MPI_Recv(out, 1, type, 0, tag, P3, status)
```

will transfer data from `in` to `out`. That is, a send to P0's first send slot is matched by a receive on P3's zeroth receive slot.

4 Manipulating Local Communicators

We now consider how the MPI interface can be extended to support LCOs. We define six new functions that are used to create a local communicator, to obtain an LCO name that can be communicated between processes, to add slots to LCOs, and to determine the number of slots associated with an LCO. Other functions can be defined to delete slots, obtain information about slots, etc., but, for brevity, we do not consider these here.

It is important to stress that operations on LCOs are local, requiring no synchronization with other processes. Thus, in particular, LCOs could not be implemented by appropriately splitting the `MPI_COMM_WORLD` communicator because this would require global synchronization.

An LCO is represented by the opaque datatype `MPI_Comm`. We will often need to be able to create an "LCO name" that can be communicated between processors, so we define the related opaque datatype `MPI_Comm_name` and the new communication datatype `MPI_CNAME`.

MPI_COMM_CREATE_LOCAL(lcomm)

OUT	lcomm	New local communicator (handle)

Create a new local communicator object, `lcomm`. Initially, no send or receive slots are associated with the new LCO. These must be added explicitly.

MPI_COMM_NAME(lcomm, name)

IN	lcomm	Local communicator object (handle)
OUT	name	Communicator name (handle)

Create and return a `name` that can be used to reference the `lcomm`. This name is used in the next function.

MPI_COMM_ADD_SEND_SLOTS(lcomm, count, lcos, slots)

INOUT	lcomm	Local communicator object (handle)
IN	count	Number of slots to add (integer ≥ 0)
IN	lcos	LCOs to be sent to (array of communicator names)
IN	slots	Slots to be sent to (array of integers)

This function and the next are used to create new connections between LCOs. This function adds `count` send slots to `lcomm` immediately after all existing send slots and defines each new slot i to be the reference to the receive slot `lcos(`i`)[slots(`i`)]`. Note that the receive slots referenced by the newly created send slots may not exist yet and must be created using MPI_COMM_ADD_RECEIVE_SLOTS. It is erroneous for a program to use a send slot, i.e., send a message on a send slot, before the target receive slot is created.

MPI_COMM_ADD_RECEIVE_SLOTS(lcomm, count)

INOUT	lcomm	Local communicator object (handle)
IN	count	Number of receive slots (integer ≥ 0)

This function adds `count` slots to the receive set of `lcomm` immediately after all existing receive slots.

MPI_COMM_NUM_SEND_SLOTS(lcomm, count)

IN	lcomm	Local communicator object (handle)
OUT	count	Number of send slots (integer ≥ 0)

Return the number of send slots in the LCO `lcomm`. Notice that if this LCO is part of a communicator structure, this function is equivalent to MPI_COMM_SIZE.

MPI_COMM_NUM_RECEIVE_SLOTS(lcomm, count)

IN	lcomm	Local communicator object (handle)
OUT	count	Number of receive slots (integer ≥ 0)

Return the number of receive slots in the LCO `lcomm`. Again, if this LCO is part of a communicator structure, this function is equivalent to MPI_COMM_SIZE.

Example: Creating a Channel. Fig. 2 creates a unidirectional channel: a pair of LCOs connected so that one can be used to send to the other. The connection is established by using an existing communicator to send a reference to one LCO (the "receive end") to the process containing the second LCO (the "send end"). A number of messages are then communicated on the channel. Notice how, at the send end, messages are sent on the single send slot, while, at the receive end, messages are received on the single receive slot.

Example: MPI_COMM_DUP. Just as MPI's point-to-point communication functions can be used to implement MPI's various global operations, so the LCO operations can be used to implement MPI's communicator functions. For example, Fig. 3 implements MPI_COMM_DUP. This function is applied collectively to a set of LCOs assumed to define an intracommunicator. `comm` is one such LCO. It constructs a new set of LCOs defining an intracommunicator with the same topology.

4.1 An Alternative Interface Design

The LCO construct defined above serves as a capability for a port, providing the ability to send or receive to or from another LCO. Once the name has been distributed, the holder of that capability is responsible for synthesizing a slot name from the port name. In situations where security or safety are issues, the ability to create a slot reference unilaterally can be problematic.

An alternative interface would associate names with specific receive slots rather than LCOs. The "add receive slots" operation then returns a *slot name*, a capability that allows another LCO to send to that receive slot. This

```
receiver_side(MPI_Comm comm, int nbr)
{
   MPI_Comm receiver;
   MPI_Comm_name rname;
   MPI_Status status;
   int msg = -1;

   /* Create an LCO representing the channel */
   MPI_Comm_create_local(&receiver);
   MPI_Comm_add_receive_slots(receiver, 1);

   /* Send LCO name to other process */
   MPI_Comm_name(receiver, &rname);
   MPI_Send(&rname, 1, MPI_CNAME, nbr, 99, comm);

   /* Receive messages from other process on channel */
   while(msg)
      MPI_Recv(&msg, 1, MPI_INT, 0, 99, receiver, &status);

   MPI_Comm_free(&receiver);
}

sender_side(MPI_Comm comm, int nbr)
{
   MPI_Comm sender;
   MPI_Comm_name rnames[1];
   MPI_Status status;
   int msg, rslots[1];

   /* Create an LCO representing the channel */
   MPI_Comm_create_local(&sender);

   /* Receive LCO name from other process,
      add to send list */
   MPI_Recv(rnames, 1, MPI_CNAME, nbr, 99, comm, &status);
   rslots[0] = 0;
   MPI_Comm_add_send_slots(sender, 1, rnames, rslots);

   /* Send messages to other process on newly created channel */
   for(msg=10; msg>=0; msg--)
      MPI_Send(&msg, 1, MPI_INT, 0, 99, sender);

   MPI_Comm_free(&sender);
}
```

Fig. 2. Implementation of a unidirectional channel using the generalized communicator constructs.

reference can be added to another LCO with a variant of the "add send slot" call with the form

MPI_COMM_ADD_SEND_SLOT(lcomm, slot-reference).

This scheme has the advantage that we can define a capability for a single receive slot rather than for the entire LCO as in the scheme described previously. A disadvantage is that, in applications that require many connections, a large number of these slot tokens must be communicated. For example, in the MPI_COMM_DUP example, $\Omega(N^2)$ slot tokens must be created and communicated, where N is the number of LCOs. In contrast, the scheme described in the preceding sections requires that only N communicator names be communicated.

Ignoring security issues, encapsulating an LCO name and a slot number into a single object (a slot name) can be a convenient user interface. For example, having such a data type permits easily sending a slot name in a message. For this reason, we implemented both interfaces in our prototype implementation described in Section 5.

```
comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
{
   int numslots, *rslots, i;
   MPI_Comm_name *names, localname;

   MPI_Comm_num_send_slots(comm, &numslots);
   names = (MPI_Comm_name *) malloc(numslots*sizeof(MPI_Comm_name));
   rslots = (int *) malloc(numslots*sizeof(int));

   /* Create our new LCO */
   MPI_Comm_create_local(newcomm);
   MPI_Comm_name(*newcomm, &localname);
   MPI_Comm_add_receive_slots(*newcomm, numslots);

   /* Gather operation collects pointers to all new LCOs */
   MPI_Allgather(&localname, 1, MPI_CNAME, names, 1, MPI_CNAME, comm);

   /* Associate these pointers with our LCO */
   for(i=0; i<numslots; i++)
      rslots[i] = i;
   MPI_Comm_add_send_slots(*newcomm, numslots, names, rslots);

   free(names);
   free(rslots);
}
```

Fig. 3. Implementation of MPI_Comm_dup using generalized communicator constructs.

4.2 Interaction with Process Groups

As noted above, MPI functions that expect a communicator as an argument behave as expected when applied to a set of LCOs that are structured so as to implement an MPI communicator. What happens when these functions are applied to LCOs that do *not* implement a communicator, either because they form less than fully connected structures or because they connect more than one communicator object per process? We propose addressing these situations by 1) generalizing the definition of existing MPI functions so that they work when applied to any LCO and 2) introducing a small number of new functions. In this article, we do not provide a detailed specification for these extensions, but instead discuss some of the issues that arise.

One issue that must be addressed relates to the fact that many MPI functions that expect a communicator as an argument are defined in terms of the *process group* associated with that communicator. For example, MPI_COMM_SIZE is defined to refer to the "number of processes in the group of comm," rather than the "number of local communicator objects." In standard MPI, these two definitions are equivalent. However, in MPI with our extensions, they are not equivalent and, in fact, we may be interested in either one or the other definition in different situations.

We address this problem by retaining the existing interpretation of any MPI function that refers explicitly to processes and by introducing new functions that operate explicitly on LCOs. To retain the existing interpretation of MPI functions that refer to processes, we provide the following definition:

Definition. *The process group associated with a local communicator object is the list of processes referenced by its send slots with duplicates removed.*

An advantage of this interpretation is that functions such as MPI_COMM_SIZE and MPI_COMM_RANK can be applied unchanged to an LCO that forms part of a communicator structure. These functions can also be applied to other LCOs, although the results may not always be useful.

Some programs will require information about LCOs rather than processes. For example, a program that creates a communicator-like structure with more LCOs than processes may want to send a message to each LCO. In this case, MPI_COMM_SIZE cannot be used to determine the number of LCOs. However, the function MPI_COMM_NUM_SEND_SLOTS provides the required information.

5 Implementation

This section describes a prototype implementation of the proposed MPI extensions. This prototype has been constructed by modifying a widely used MPI implementation, MPICH [13]. The modifications required for any MPI implementation are inevitably focused within the MPI communicator construct. Hence, we begin by describing how communicators are represented within MPICH.

The two principal components of an MPI communicator as represented in MPICH are a process group and a context. The process group is represented by a sequence of process identifiers stored as an integer array. A process's rank in a group refers to its index in this array. The array contains for each index an address in a format that the underlying device can use and understand, for example, the rank in MPI_COMM_WORLD. The context associated with a communicator is represented by an integer. Note that the communicator data structure maintained in each process has the same process group and context values. These were determined by the collective operation that created the communicator. When a message is sent, the rank provided in the send call is used to extract a process identifier from the process group array associated with the communicator on which the send is performed. The message is then sent to that process, together with a message envelope containing the rank of the sending process, the tag, and the integer context identifier associated with the communicator.

An LCO has a somewhat different structure. Corresponding to the MPICH integer representation of a context is an integer LCO identifier, which is assigned when the LCO is created. This identifier is guaranteed to be unique only within the creating process. Corresponding to the MPICH process group is an array of send slots. Each entry in an LCO's send-slot array contains a process identifier, an LCO identifier, and a receive-slot index. Receive operations proceed in a manner identical to an MPI receive. A send operation differs from an MPI send only in that, when constructing the message envelope, it uses the receive slot index for the rank and the LCO identifier as the context. We note that one significant advantage of this approach relative to the MPICH communicator structure is that identifiers can be allocated in a purely local fashion. Hence, collective operations are not required for communicator creation and the identifier name space can be more densely populated.

The execution time overhead introduced by this modification is minimal: When sending a message, two additional array references are required to determine the appropriate context and receive slot and, when receiving a message, no additional code is required. The principal overhead is thus the additional space required to maintain an LCO identifier and receive-slot index in each send slot, although this is still a constant factor times the number of send slots. In addition, one can imagine optimizations that recognize sets of LCOs representing MPI intracommunicator or intercommunicator structures and revert to the more compact representation in this case.

In summary, the required changes to MPICH were minimal and induce almost no overhead. It would be almost impossible to experimentally measure the few additional instructions added in our implementation. Hence, the generalized communicator extensions are practical and easy to implement, while greatly increasing the flexibility of MPI communicators and MPI-2 dynamic process spawning.

An alternative implementation approach would use a communication library, such as Nexus [14], that provides global pointer and single-sided communication operations. In this environment, a send slot can be represented as a global pointer to a remote queue corresponding to a receive slot and a send operation can be implemented as a remote enqueue operation. This technique has been used to construct an implementation of ordinary MPI [15].

6 Conclusions

We have presented extensions to the MPI communicator that permit the representation of more general and flexible communication structures. These extensions are backwards compatible with MPI, meaning that any existing MPI program will execute correctly in a system that supports the new constructs. We believe that the new constructs can be incorporated into existing MPI implementations without difficulty and without significant performance degradation.

A disadvantage of the extensions as presented here is that, because LCOs (and slots within LCOs) are created and destroyed independently, we lose MPI's message safety property. That is, a message may arrive for a nonexistent receive slot. This problem can be avoided, at the expense of added complexity, by using one of the various mechanisms that have been developed for managing distributed objects, such as reference counting.

The generalized LCO proposed in this article also appears to have other uses. For example, LCOs can be used to manage "one-sided" communications in which the arrival of a message triggers the execution of a handler function. By requiring these communications to occur over an LCO, we provide an endpoint on the receiver side with which control information can be associated. LCOs can also be used to define generalized collective communication operations in which user-defined transformations are applied to data supplied by an arbitrary number of senders and the results of these transformations are delivered to an arbitrary number of receivers.

ACKNOWLEDGMENTS

This work was supported by the Natural Science and Engineering Research Council of Canada, the US National Science Foundation's Center for Research in Parallel Computation, under Contract CCR-8809615, and by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, US Department of Energy, under Contract W-31-109-Eng-38.

REFERENCES

[1] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface.* MIT Press, 1994.

[2] M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, and J. Dongarra, *MPI: The Complete Reference.* MIT Press, 1996.

[3] J.M. Squyres, B.C. McCandless, and A. Lumsdaine, "Object Oriented MP: A Class Library for the Message Passing Interface," *Proc. Parallel Object-Oriented Methods and Applications Conf.,* http://www.mpi.nd.edu/research/oompi/documentation.php, 1996.

[4] "MPI-2: Extensions to the Message-Passing Interface," Message Passing Interface Forum, http://www.mpi-forum.org, 1997.

[5] J.H. Reppy, "Concurrent ML: Design, Application, and Semantics," *Functional Programming, Concurrency, Simulation, and Automated Reasoning,* 1993.

[6] I. Foster and K.M. Chandy, "Fortran M: A Language for Modular Parallel Programming," *J. Parallel and Distributed Computing,* 1994.

[7] I. Foster and S. Taylor, "A Compiler Approach to Scalable Concurrent Program Design," *ACM Trans. Programming Languages and Systems,* vol. 16, no. 3, pp. 577-604, 1994.

[8] I. Foster and S. Taylor, *Strand: New Concepts in Parallel Programming.* Prentice Hall, 1990.

[9] A. Skjellum, N. Doss, K. Viswanathan, A. Chowdappa, and P. Bangalore, "Extending the Message Passing Interface," *Proc. 1994 Scalable Parallel Libraries Conf.,* 1994.

[10] "Document for the Real-Time Message Passing Interface (MPI/RT-1. 0)," Real-Time Message Passing Interface Forum, http://www.mpirt.org, 2000.

[11] I. Foster, C. Kesselman, and M. Snir, "Generalized Communicators in the Message Passing Interface," *Proc. 1996 MPI Developers Conf.,* pp. 42-49, 1996.

[12] M. Haines, P. Mehrotra, and D. Cronk, "Ropes: Support for Collective Operations among Distributed Threads," Technical Report 95-36, Inst. for Computer Application in Science and Eng., 1995.

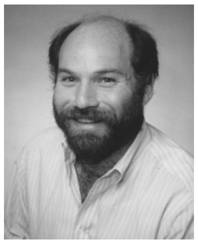
[13] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard," *Parallel Computing,* vol. 22, pp. 789-828, 1996.

[14] I. Foster, C. Kesselman, and S. Tuecke, "The Nexus Approach to Integrating Multithreading and Communication," *J. Parallel and Distributed Computing,* vol. 37, pp. 70-82, 1996.

[15] I. Foster, J. Geisler, W. Gropp, N. Karonis, E. Lusk, G. Thiruvathukal, and S. Tuecke, "A Wide-Area Implementation of the Message Passing Interface," *Parallel Computing,* vol. 24, no. 11, 1998.

Erik Demaine received the MMath degree from the University of Waterloo and the BSc degree from Dalhousie University. He is currently a PhD student in computer science at the University of Waterloo. His research interests include parallel and distributed computing, particularly message passing, as well as algorithms, data structures, and computational geometry. He is a member of the IEEE Computer Society.

Ian Foster is a senior scientist and associate director of the Mathematics and Computer Science Division at Argonne National Laboratory. He is a professor of computer science at the University of Chicago and a senior fellow in the Argonne/U.Chicago Computation Institute. He has published four books and many papers and technical reports in parallel and distributed processing, software engineering, and computational science. He currently coleads the Globus project with Dr. Carl Kesselman of USC/ISI, which was awarded the 1997 Global Information Infrastructure "Next Generation" award and which provides protocols and services used by many distributed computing projects worldwide. He cofounded the influential Grid Forum and recently coedited a book on this topic, published by Morgan-Kaufmann, entitled *The Grid: Blueprint for a New Computing Infrastructure*.



Carl Kesselman received the BS degree in electrical engineering from the State University of New York at Buffalo, the MS degree in electrical engineering from the University of Southern California, and the PhD degree in computer science from the University of California at Los Angles. He is the director of the Center for Grid Technologies at the University of Southern California's Information Sciences Institute. He is also a research associate professor of computer science at the University of Southern California and a visiting associate in computer science at the California Institute of Technology. His research interests are focused on all aspects of grid computing, including grid architecture, wide-area data management services, resource management, and security. Along with Dr. Ian Foster, Dr. Kesselman coleads the Globus project, which is developing core technologies for grid systems. He is a member of the IEEE and the Computer Society

Marc Snir received the PhD degree in mathematics from the Hebrew University of Jerusalem in 1979. He is senior manager at the IBM T.J. Watson Research Center, where he leads research on the Blue Gene massively parallel system. He worked at New York University (NYU) on the NYU Ultracomputer project from 1980 to 1982 and worked at the Hebrew University of Jerusalem from 1982 to 1986, when he joined the IBM T.J. Watson Research Center. At IBM, he headed research that led to the line of high performance IBM SP systems. He has published more than 100 journal and conference papers on computational complexity, parallel algorithms, parallel architectures, and parallel programming. He also coauthored the MPI standard and contributed to other languages, libraries, and tools for parallel programming. He is on the editorial board of *Transactions on Computer Systems and Parallel Processing Letters*. He is member of the IBM Academy of Technology, a fellow of the ACM, a fellow of the IEEE, and a member of the IEEE Computer Society.