MPI: A STANDARD MESSAGE PASSING INTERFACE[1]

David W. Walker
Mathematical Sciences Section
Oak Ridge National Laboratory
P. O. Box 2008, Bldg. 6012
Oak Ridge, TN 37831-6367

Jack J. Dongarra
Mathematical Sciences Section
Oak Ridge National Laboratory
P. O. Box 2008, Bldg. 6012
Oak Ridge, TN 37831-6367
and
Department of Computer Science
University of Tennessee
Knoxville, TN 37996-1301

[1]This work was supported in part by ARPA under contract number DAAL03-91-C-0047 administered by ARO, and in part by DOE under contract number DE-AC05-84OR21400.

MPI: A STANDARD MESSAGE PASSING INTERFACE

David W. Walker and Jack J. Dongarra

MPI is a proposed standard message passing interface whose use on massively parallel computers and networks of workstations is becoming widespread. The design of MPI was a collective effort involving researchers in the United States and Europe from many organizations and institutions. MPI includes point-to-point and collective communication routines, as well as support for general datatypes, application topologies, and the construction of safe and modular parallel application libraries through the communicator abstraction. While making use of new ideas where appropriate, the MPI standard is based largely on current practice.

1 Introduction

MPI is a proposed standard message passing interface originally designed for writing applications and libraries for distributed memory environments. The main advantages of establishing a message passing interface for such environments are portability and ease-of-use, and a standard message passing interface is a key component in building a concurrent computing environment in which applications, software libraries, and tools can be transparently ported between different machines. Furthermore, the definition of a message passing standard provides vendors with a clearly defined set of routines that they can implement efficiently, or in some cases provide hardware or low-level system support for, thereby enhancing scalability.

The functionality that MPI is designed to provide is based on current common practice, and is similar to that provided by widely-used message passing systems such as Express [11], NX/2 [12], Vertex, [10], PARMACS [7, 8], and P4 [9]. In addition, the flexibility and usefulness of MPI has been broadened by incorporating ideas from more recent and innovative message passing systems such as CHIMP [4, 5], Zipcode [13, 14], and the IBM External User Interface [6].

The MPI standardization effort involved about 60 people from 40 organizations mainly from the United States and Europe. Most of the major vendors of concurrent computers were involved in MPI, along with researchers from universities, government laboratories, and industry. The standardization process began with the Workshop on Standards for Message Passing in a Distributed Memory Environment, sponsored by the Center for Research on Parallel Computing, held April 29-30, 1992, in Williamsburg, Virginia [15]. At this workshop the basic features essential to a standard message passing interface were

discussed, and a working group established to continue the standardization process.

A preliminary draft proposal, now known as MPI-0, was put forward by Dongarra, Hempel, Hey, and Walker in November 1992, and a revised version was completed in February 1993 [3]. MPI-0 embodies the main features that were identified at the Williamsburg workshop as being necessary in a message passing standard. This proposal was intended to initiate discussion of standardization issues within the distributed memory concurrent computing community, and has served as a basis for the subsequent MPI standardization process. Although MPI-0 and the MPI draft standard described here have many features in common, they are distinct proposals with the latter being broader in scope and paying greater attention to important issues such as thread safety. In November 1992, the MPI working group was superseded by the establishment of the MPI Forum. This group, membership of which was open to all those involved in high performance computing, met regularly throughout the first nine months of 1993 to produce the first draft of the MPI specification which was presented at the Supercomputing '93 conference in November 1993. After a period of public review and comment, version 1.0 of the MPI specification was released in May 1994. Minor clarifications and corrections were made to this document over the following twelve months and a revised version of the specification, version 1.1, was released in June 1995. The specification of MPI as set forth in version 1.1 is generally known as MPI-1 to distinguish it from MPI-0 and an on-going effort to extend the specification known as MPI-2. All mention of MPI in this article refers to MPI-1, unless explicitly stated otherwise.

2 An Overview of MPI

MPI is intended to be a standard message passing interface for applications and libraries running on concurrent computers with a logically distributed memory. MPI is not specifically designed for use by parallelizing compilers. MPI provides no explicit support for multithreading, although one of the design goals of MPI was to ensure that it can be implemented efficiently in a multithreaded environment. The MPI standard does not mandate that an implementation should be interoperable with other MPI implementations. However, MPI does provide to message passing routines all the information needed to allow a single MPI implementation to operate in a heterogeneous environment.

The main features of MPI are as follows:

- A set of routines that support point-to-point communication between pairs of processes. Blocking and nonblocking versions of the routines are provided which may be used in four different *communication modes*. As discussed in Section 4.1, these modes correspond to different communication protocols. Message selectivity in point-to-point communication is

by source process and message tag, each of which may be wildcarded to indicate that any valid value is acceptable.

- The communicator abstraction that provides support for the design of safe, modular parallel software libraries.
- General, or derived, datatypes, that permit the specification of messages of noncontiguous data of differing datatypes.
- Application topologies that specify the logical layout of processes. A common example is a Cartesian grid which is often used in two and three-dimensional problems.
- A rich set of collective communication routines that perform coordinated communication among a set of processes.

In MPI there is no mechanism for creating processes, and an MPI program is parallel *ab initio*, i.e., there is a fixed number of processes from the start to the end of an application program. All processes are members of at least one process group. Initially all processes are members of the same group, and a number of routines are provided that allow an application to create (and destroy) new subgroups. Within a group each process is assigned a unique rank in the range 0 to $n-1$, where n is the number of processes in the group. This rank is used to identify a process, and, in particular, is used to specify the source and destination processes in a point-to-point communication operation, and the root process in certain collective communication operations.

MPI was designed as a message passing interface rather than a complete parallel programming environment, and in its current form intentionally omits many desirable features. For example, MPI lacks

- mechanisms for process creation and control;
- one-sided communication operations that would permit put and get messages, and active messages;
- nonblocking collective communication operations, and the ability for a collective communication operation to involve more than one group;
- language bindings for Fortran 90 and C++.

These issues, and other possible extensions to MPI, are currently being considered in the MPI-2 effort. Extensions to MPI for performing parallel I/O are also under consideration.

3 The Communicator Abstraction

Communicators provide support for the design of safe, modular software libraries. Here "safe" means that messages intended for receipt by a particular receive call in an application will not be incorrectly intercepted by a different receive call. Thus, communicators are a powerful mechanism for avoiding unintentional non-determinism in message passing. This may be a particular problem when using third-party software libraries that perform message passing. The point here is that the application developer has no way of knowing if the tag, group, and rank completely disambiguate the message traffic of different libraries and the rest of the application. Communicator arguments are passed to all MPI message passing routines, and a message can be communicated only if the communicator arguments passed to the send and receive routines match. Thus, in effect communicators provide an additional criterion for message selection, and hence permit the construction of independent tag spaces.

If communicators are not used to disambiguate message traffic there are two ways in which a call to a library routine can lead to unintended behavior. In the first case the processes enter a library routine synchronously when a send has been initiated for which the matching receive is not posted until after the library call. In this case the message may be incorrectly received in the library routine. The second possibility arises when different processes enter a library routine asynchronously, as shown in the example in Figure 1, resulting in nondeterministic behavior. If the program behaves correctly processes 0 and 1 each receive a message from process 2, using a wildcarded selection criterion to indicate that they are prepared to receive a message from any process. The three processes then pass data around in a ring within the library routine.

If separate communicators are not used for the communication inside and outside of the library routine this program may intermittently fail. Suppose we delay the sending of the second message sent by process 2, for example, by inserting some computation, as shown in Figure 2. In this case the wildcarded receive in process 0 is satisfied by a message sent from process 1, rather than from process 2, and deadlock results. By supplying a different communicator to the library routine we can ensure that the program is executed correctly, regardless of when the processes enter the library routine.

Communicators are opaque objects, which means they can only be manipulated using MPI routines. The key point about communicators is that when a communicator is created by an MPI routine it is guaranteed to be unique. Thus it is possible to create a communicator and pass it to a software library for use in all that library's message passing. Provided that communicator is not used for any message passing outside of the library, the library's messages and those of the rest of the application cannot be confused.

Communicators have a number of attributes. The group attribute identifies the process group relative to which process ranks are interpreted, and/or which identifies the process group involved in a collective communication operation.

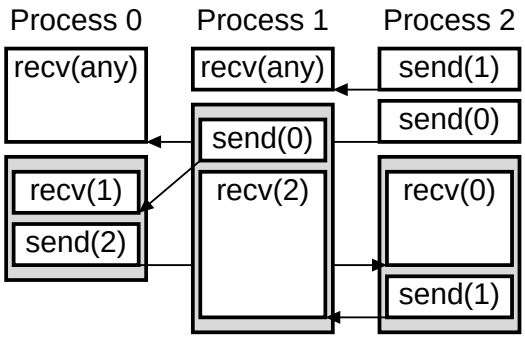

Figure 1: Use of communicators. Time increases down the page. Numbers in parentheses indicate the process to which data are being sent or received. The gray shaded area represents the library routine call. In this case the program behaves as intended. Note that the second message sent by process 2 is received by process 0, and that the message sent by process 0 is received by process 2.

Communicators also have a topology attribute which gives the topology of the process group. Topologies are discussed in Section 6. In addition, users may associate arbitrary attributes with communicators through a mechanism known as caching.

4 Point-To-Point Communication

MPI provides routines for sending and receiving blocking and nonblocking messages. A blocking send does not return until it is safe for the application to alter the message buffer on the sending process without corrupting or changing the message sent. A nonblocking send may return while the message buffer on the sending process is still volatile, and it should not be changed until it is guaranteed that this will not corrupt the message. This may be done by either calling a routine that blocks until the message buffer may be safely reused, or by calling a routine that performs a nonblocking check on the message status. A blocking receive suspends execution on the receiving process until the incoming message has been placed in the specified application buffer. A nonblocking receive may return before the message has been received into the specified application buffer, and a subsequent call must be made to ensure that this has occurred before the application uses the data in the message.

4.1 Communication Modes

In MPI a message may be sent in one of four communication modes, which approximately correspond to the most common protocols used for point-to-point

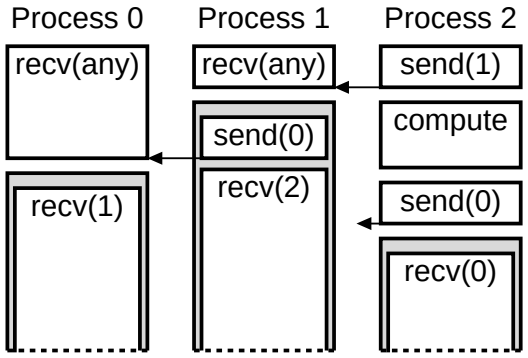

Figure 2: Unintended behavior of program. In this case the message from process 2to process 0 is never received, and deadlock results.

communication. In *ready* mode a message may be sent only if a corresponding receive has been initiated. In *standard* mode a message may be sent regardless of whether a corresponding receive has been initiated. MPI includes a *synchronous* mode which is the same as the standard mode, except that the send operation will not complete until a corresponding receive has been initiated on the destination process. Finally, there is a *buffered* mode. To use buffered mode the user must first supply a buffer and associate it with a communicator. When a subsequent send is performed using that communicator MPI may use the associated buffer to buffer the message. A buffered send may be performed regardless of whether a corresponding receive has been initiated.

There are, therefore, 8 types of send operation and 2 types of receive, as shown in Figure 3. In addition, routines are provided that send to one process while receiving from another. Different versions are provided for when the send and receive buffers are distinct, and for when they are the same. The send/receive operation is blocking, so does not return until the send buffer is ready for reuse, and the incoming message has been received. The two send/receive routines bring the total number of point-to-point message passing routines up to 12.

4.2 Persistent Communication Requests

MPI also provides a set of routines for creating communication request objects that completely describe a send or receive operation by binding together all the parameters of the operation. A handle to the communication object so formed is returned, and may be passed to a routine that actually initiates the communication. As with the nonblocking communication routines in Fig. 3, a subsequent call should be made to ensure completion of the operation.

Persistent communication objects may be used to optimize communication

SEND	Blocking	Nonblocking
Standard	`mpi_send`	`mpi_isend`
Ready	`mpi_rsend`	`mpi_irsend`
Synchronous	`mpi_ssend`	`mpi_issend`
Buffered	`mpi_bsend`	`mpi_ibsend`

RECEIVE	Blocking	Nonblocking
Standard	`mpi_recv`	`mpi_irecv`

Figure 3: Classification and names of the point-to-point send and receive routines.

performance, particularly when the same communication pattern is repeated many times in an application. For example, if a send routine is called within a loop, performance may be improved by creating a communication request object that describes the parameters of the send prior to entering the loop, and then initiating the communication inside the loop to send the data on each pass through the loop.

There are five routines for creating communication objects: four for send operations (one corresponding to each communication mode), and one for receive operations. A persistent communication object should be deallocated when no longer needed.

5 General Datatypes

All point-to-point message passing routines in MPI take as an argument the datatype of the data communicated. In the simplest case this will be a primitive datatype, such as an integer or floating point number. However, MPI also supports more general datatypes, and thereby supports the communication of array sections and structures involving combinations of primitive datatypes.

A general datatype is a sequence of pairs of primitive datatypes and integer byte displacements. Thus,

$$\text{Datatype} = \{ (\text{type}_0, \text{disp}_0), (\text{type}_1, \text{disp}_1), \ldots, (\text{type}_{n-1}, \text{disp}_{n-1}) \}$$

Together with a base address, a datatype specifies a communication buffer. General datatypes are built up hierarchically from simpler components. There are four basic constructors for datatypes, namely the contiguous, vector, indexed, and struct constructors. We will now discuss each of these in turn.

The *contiguous* constructor creates a new datatype from repetitions of a specified old datatype. This requires us to specify the old datatype and the

number of repetitions, n. For example, if the old datatype is `oldtype` = { (`double`, 0), (`char`, 8) } and $n = 3$, then the new datatype would be,

{ (`double`, 0), (`char`, 8), (`double`, 16), (`char`, 24), (`double`, 32), (`char`, 40) }

It should be noted how each repeated unit in the new datatype is aligned with a double word boundary. This alignment is dictated by the appearance of a `double` in the old datatype, so that the extent of the old datatype is taken as 16 bytes, rather than 9 bytes.

The *vector* constructor builds a new datatype by replicating an old datatype in blocks at fixed offsets. The new datatype consists of `count` blocks, each of which is a repetition of `blocklen` items of some specified old datatype. The starts of successive blocks are offset by `stride` items of the old datatype. Thus, if `count` = 2, `blocklen` = 3, and `stride` = 4 then the new datatype would be,

{ (`double`, 0), (`char`, 8), (`double`, 16), (`char`, 24), (`double`, 32), (`char`, 40),
(`double`, 64), (`char`, 72), (`double`, 80), (`char`, 88), (`double`, 96), (`char`, 104)}

Here the offset between the two blocks is 64 bytes, which is the stride multiplied by the extent of the old datatype.

The *indexed* constructor is a generalization of the vector constructor in which each block has a different size and offset. The sizes and offsets are given by the entries in two integer arrays, `B` and `I`. The new datatype consists of `count` blocks, and the ith block is of length `B[i]` items of the specified old datatype. The offset of the start of the ith block is `I[i]` items of the old datatype. Thus, if `count` = 2, `B` = {3, 1}, and `I` = {64, 0}, then the new datatype would be,

{ (`double`, 64), (`char`, 72), (`double`, 80), (`char`, 88), (`double`, 96), (`char`, 104),
(`double`, 0), (`char`, 8) }

The *struct* constructor is the most general of the datatype constructors. This constructor generalizes the indexed constructor by allowing each block to be of a different datatype. Thus, in addition to specifying the number of blocks, `count`, and the block length and offset arrays, `B` and `I`, we must also give the datatype of the replicated unit in each block. Let us assume this is specified in an array `T`. The length of the ith block is `B[i]` items of type `T[i]`, and the offset of the start of the ith block is `I[i]` bytes. Thus, if `count=3`, `T` = {`MPI_Float`, `oldtype`, `MPI_Char`}, `I` = {0, 16, 26}, and `B` = {2, 1, 3}, then the new datatype would be,

{ (`float`, 0), (`float`, 4), (`double`, 16), (`char`, 24), (`char`, 26) (`char`, 27) (`char`, 28) }

In addition to the constructors described above, there is a variant of the vector constructor in which the stride is given in bytes instead of the number of items. There is also a variant of the indexed constructor in which the block offsets are given in bytes.

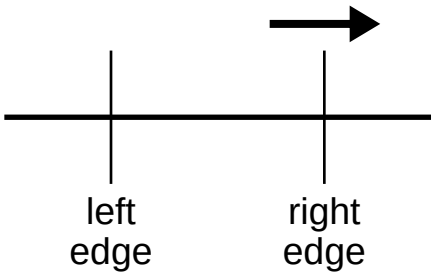

Figure 4: Particle migration in a one-dimensional code. The left and right edges of a process domain are shown. We shall consider just the migration of particles across the righthand boundary.

To better understand the use of general data structures consider the example of an application in which particles move on a one-dimensional domain. We assume that each process is responsible for a different section of this domain. In each time step particles may move from the subdomain of one process to that of another, and so the data for such particles must be communicated between processes. We shall just consider here the task of migrating particles across the righthand boundary of each process, as shown in Figure 4. The particle data are stored in an array of structures, with each entry in this structure consisting of the particle position, `x`, velocity, `v`, and type, `k`:

```
struct Pstruct { double x; double v; int k; };
```

The C code for migrating particles across the righthand boundary is shown in Figure 5.

In Figure 5 the code in the first box creates a datatype, `Ptype`, that represents the `Pstruct` structure for a single particle. This datatype is,

```
Ptype = {(double,0), (double,8), (int,16)}
```

In the second code box the particles that have crossed the righthand boundary are identified, and their index in the `particle` array is stored in `Pindex`. It is assumed that no more than 100 particles cross the boundary. The call to `MPI_Type_indexed` uses an indexed constructor to create a new datatype, `Ztype`, that references all the migrating particles. Before communicating the data, the `Ztype` datatype must be committed. This is done to allow the system to use a different internal representation for `Ztype`, and to optimize the communication operation. Committing a datatype is most likely to be advantageous when reusing a datatype many times, which is not the case in this example. Finally, the migrating particles are communicated by a call to `MPI_Sendrecv`. The offsets in the `Ztype` datatype are interpreted relative to the address of the start of the `particle` array.

```
struct Pstruct particle[1000], recvbuf[100];
MPI_Datatype Ptype, Ztype;
MPI_Datatype Stype[3]={MPI_Double, MPI_Double, MPI_Int};
MPI_Aint Sindex[3];
int Sblock[3]={1, 1, 1};
int Pindex[100];
int Pblock[100];

Sindex[0] = 0;
Sindex[1] = sizeof(double);
Sindex[2] = 2*sizeof(double);
MPI_Type_struct (3, Sblock, Sindex, Stype, &Ptype);

j=0;
for (i=0;i<1000;i++)
  if (particle[i].x > right_edge) {
    Pindex[j] = i;
    Pblock[j] = 1;
    j++;}
MPI_Type_indexed (j, Pblock, Pindex, Ptype, &Ztype);

MPI_Type_commit (Ztype);
MPI_Sendrecv (particle, 1, Ztype, dest, tag,
          recvbuf, 100, Ptype, source, tag,
          comm, &status);
```

Figure 5: Fragment of C code for migrating particles across the righthand process boundary. Outgoing particles are sent to process `dest` using the derived datatype, `Ztype`. Incoming particles from process `source` are received into the particle array `recvbuf` which has enough space for 100 particles.

6 Application Topologies

In many applications the processes are arranged with a particular topology, such as a two- or three-dimensional grid. MPI provides support for general application topologies that are specified by a graph in which processes that communicate a significant amount are connected by an arc. If the application topology is an n-dimensional Cartesian grid then this generality is not needed, so as a convenience MPI provides explicit support for such topologies. For a Cartesian grid, periodic or nonperiodic boundary conditions may apply in any specified grid dimension. In MPI, a group either has a Cartesian or graph topology, or no topology. In addition to providing routines for translating between process rank and location in the topology, MPI also:

1. allows knowledge of the application topology to be exploited in order to efficiently assign processes to physical processors,

2. provides a routine for partitioning a Cartesian grid into hyperplane groups by removing a specified set of dimensions,

3. provides support for shifting data along a specified dimension of a Cartesian grid.

By dividing a Cartesian grid into hyperplane groups it is possible to perform collective communication operations within these groups. In particular, if all but one dimension is removed a set of one-dimensional subgroups is formed, and it is possible, for example, to perform a multicast in the corresponding direction.

7 Collective Communication

Collective communication routines provide for coordinated communication among a group of processes [1, 2]. The process group is given by the communicator object that is input to the routine. The MPI collective communication routines have been designed so that their syntax and semantics are consistent with those of the point-to-point routines. The collective communication routines maybe (but do not have to be) implemented using the MPI point-to-point routines. Collective communication routines do not have message tag arguments, though an implementation in terms of the point-to-point routines may need to make use of tags. A collective communication routine must be called by all members of the group with consistent arguments. As soon as a process has completed its role in the collective communication it may continue with other tasks. Thus, a collective communication is not necessarily a barrier synchronization for the group. MPI does not include nonblocking forms of the collective communication routines. MPI collective communication routines are divided into two broad classes: data movement routines, and global computation routines.

7.1 Collective Data Movement Routines

There are five basic types of collective data movement routine: broadcast, scatter, and gather, all-gather, and all-to-all. These are illustrated in Figure 6.

The broadcast routine broadcasts data from one process to all other processes in the group. The scatter routine sends distinct data from one process to all processes in the group. This is also known as "one-to-all personalized communication". In the gather routine one process receives input from each process and concatenates it in rank order. The all-gather routine broadcasts data from each process to all others, and on completion each has received the same data. Thus, for the all-gather routine each process ends up with the same output data, which is the concatenation of the input data of all processes, in rank order. In the all-to-all routine each process scatters distinct data to all processes in the group, so the processes receive different data from each process. This is also known as "all-to-all personalized communication".

In addition, MPI provides vector versions of all these 5 routines, except the one-all broadcast, in which each process can send and/or receive a different number of data items to and/or from each process.

7.2 Global Computation Routines

There are two global computation routines in MPI: reduce and scan. Different versions of the reduction routine are provided depending on whether the results are made available to all processes in the group, just one process, or are scattered cyclicly across the group. Common reduction operations are the evaluation of the maximum, minimum, or sum of a set of values distributed across a group of processes. The scan routines perform a parallel prefix with respect to a user-specified operation on data distributed across a specified group. If D_i is the data item on the process with rank i, then on completion the output buffer of this process contains the result of combining the values from the processes with rank $0, 1, \ldots, i$, i.e.,

$$\mathcal{D}_i = D_0 \oplus D_1 \oplus D_2 \oplus \cdots \oplus D_i \tag{1}$$

MPI provides a set of pre-defined functions, and a mechanism through which user-specified functions can be used in reduce and scan operations.

8 Summary

This paper has given an overview of the main features of MPI, but has not described the detailed syntax of the MPI routines, or discussed language binding issues. These are fully discussed in the MPI specification document. Since its first release in May 1994, MPI has rapidly become a popular message passing interface for parallel application programs and libraries. Several portable implementations of MPI exist, and major vendors are developing optimized versions for

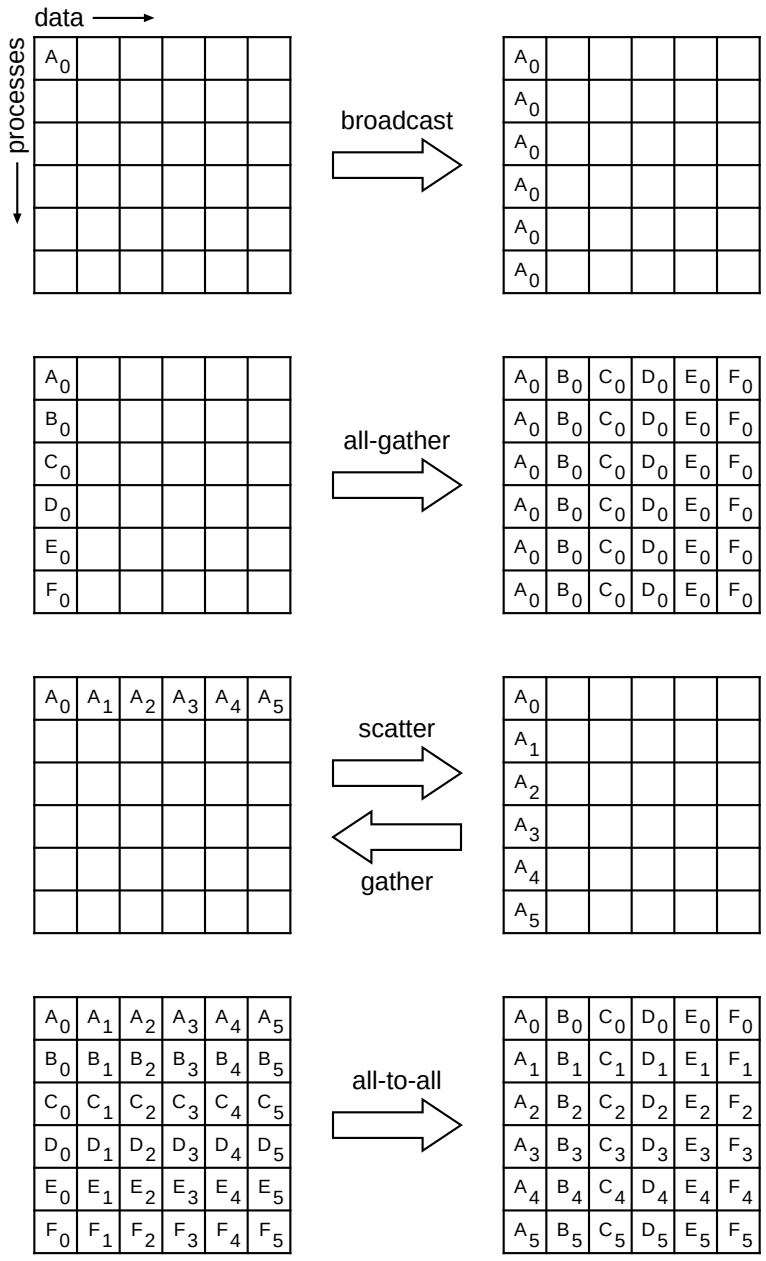

Figure 6: Basic collective data movement routines for a group of six processes. In each case, each row of boxes represents data locations in one process. Thus, in the broadcast routine, initially just the first process contains the data A_0, but after the broadcast all processes contain it.

their own machines. The MPI Forum is currently considering extensions to MPI that will significantly expand its usefulness. The specification for these extensions is expected to be released in 1997. Further information about MPI, including the specification document and information about the portable implementations, may be obtained from the World-Wide Web. Two good starting points are `http://www.epm.ornl.gov/~walker/` and `http://www.mcs.anl.gov/mpi`.

References

[1] V. Bala, J. Bruck, R. Cypher, P. Elustondo, A. Ho, C.-T. Ho, S. Kipnis, and Marc Snir. Ccl: A portable and tunable collective communication library for scalable parallel computers. Technical report, IBM T. J. Watson Research Center, 1993. Preprint.

[2] J. Bruck, R. Cypher, P. Elustondo, A. Ho, C.-T. Ho, S. Kipnis, and Marc Snir. Ccl: A portable and tunable collective communication library for scalable parallel computers. Technical report, IBM Almaden Research Center, 1993. Preprint.

[3] J. J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker. A proposal for a user-level, message passing interface in a distributed memory environment. Technical Report TM-12231, Oak Ridge National Laboratory, February 1993.

[4] Edinburgh Parallel Computing Centre, University of Edinburgh. *CHIMP Concepts*, June 1991.

[5] Edinburgh Parallel Computing Centre, University of Edinburgh. *CHIMP Version 1.0 Interface*, May 1992.

[6] D. Frye, R. Bryant, H. Ho, R. Lawrence, and M. Snir. An external user interface for scalable parallel systems. Technical report, IBM, May 1992.

[7] R. Hempel. The ANL/GMD macros (PARMACS) in fortran for portable parallel programming using the message passing programming model – users' guide and reference manual. Technical report, GMD, Postfach 1316, D-5205 Sankt Augustin 1, Germany, November 1991.

[8] R. Hempel, H.-C. Hoppe, and A. Supalov. A proposal for a PARMACS library interface. Technical report, GMD, Postfach 1316, D-5205 Sankt Augustin 1, Germany, October 1992.

[9] Ewing Lusk, Ross Overbeek, et al. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., 1987.

[10] nCUBE Corporation. *nCUBE 2 Programmers Guide, r2.0*, December 1990.

[11] Parasoft Corporation. *Express Version 1.0: A Communication Environment for Parallel Computers*, 1988.

[12] Paul Pierce. The NX/2 operating system. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 384–390. ACM Press, 1988.

[13] A. Skjellum and A. Leung. Zipcode: a portable multicomputer communication library atop the reactive kernel. In D. W. Walker and Q. F. Stout, editors, *Proceedings of the Fifth Distributed Memory Concurrent Computing Conference*, pages 767–776. IEEE Press, 1990.

[14] A. Skjellum, S. Smith, C. Still, A. Leung, and M. Morari. The Zipcode message passing system. Technical report, Lawrence Livermore National Laboratory, September 1992.

[15] D. Walker. Standards for message passing in a distributed memory environment. Technical Report TM-12147, Oak Ridge National Laboratory, August 1992.