An Introduction to the MPI Standard

Jack J. Dongarra
University of Tennessee and Oak Ridge National Laboratory
Steve W. Otto
Oregon Graduate Institute of Science & Technology
Marc Snir
IBM, T.J. Watson Research Center
David Walker
Oak Ridge National Laboratory

October 15, 1995

Contents

1 Introduction **3**

2 Overview **4**

3 Goals **4**

4 What MPI Does and Does Not Specify **5**

5 Point to Point Communication **6**

6 User-defined Datatypes **8**

7 Collective Communications **9**

8 Groups, Contexts, and Communicators **11**

9 Conclusions **12**

A Sidebar: Implementations of MPI **15**

B Sidebar: More Information on MPI, Assistance **16**

C Sidebar: Library Communicator and Caching **17**

1 Introduction

The Message Passing Interface (MPI) is a portable message-passing standard that facilitates the development of parallel applications and libraries. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in Fortran 77 or C. MPI also forms a possible target for compilers of languages such as High Performance Fortran [8]. Commercial and free, public-domain implementations of MPI already exist (see sidebar A). These run on both tightly-coupled, massively-parallel machines (MPPs), and on networks of workstations (NOWs).

The MPI standard was developed over a year of intensive meetings and involved over 80 people from approximately 40 organizations, mainly from the United States and Europe. Meeting attendance was open to the technical community. The meets where announced on various bulletin boards and mailing lists. MPI operated on a very tight budget (in reality, it had no budget when the first meeting was announced). The Advanced Research Projects Agency (ARPA) through the National Science Foundation (NSF) have provided partial travel support for the U.S. academic participants. Support for several European participants was provided by the European Commission through the ESPRIT project. Formal voting at the meetings was done by a single vote per organization. In order to vote, the organization must have had a representative at two of the last three meetings. In order to give guidance for preparation of formal proposals, informal votes where often taken involving everyone present. Many vendors of concurrent computers were involved, along with researchers from universities, government laboratories, and industry. This effort culminated in the publication of the MPI specification [5]. Other sources of information on MPI are available [9] or are under development (see sidebar B).

Researchers incorporated into MPI the most useful features of several systems, rather than choosing one system to adopt as the standard. MPI has roots in PVM [3, 6], Express [10], P4 [1], Zipcode [11], and Parmacs [2], and in systems sold by IBM, Intel, Meiko, Cray Research, and Ncube.

2 Overview

MPI is used to specify the communication among a set of processes forming a concurrent program. The message-passing paradigm is attractive because of its wide portability and scalability. It is easily compatible with both distributed-memory multicomputers and shared-memory multiprocessors, NOWs, and combinations of these elements. Message passing will not be made obsolete by increases in network speeds or by architectures combining shared and distributed-memory components.

Though much of MPI serves to standardize the "common practice" of existing systems, MPI has gone further and defined advanced features such as user-defined datatypes, persistent communication ports, powerful collective communication operations, and scoping mechanisms for communication. No previous system incorporated all these features.

3 Goals

In considering MPI, it is important to understand the goals of the standardization effort, the constraints such an endeavor implies, and the practical constraints under which the committee operated. Some of these are listed below.

- Design an application programming interface (not necessarily for compilers or a system implementation library).
- Allow efficient communication: Avoid memory-to-memory copying and allow overlap of computation and communication and offload to communication co-processor, where available.
- Allow for implementations that can be used in a heterogeneous environment.
- Allow convenient C and Fortran 77 bindings for the interface.
- Assume a reliable communication interface: the user need not cope with communication failures. Such failures are dealt with by the underlying communication subsystem.

- Define an interface that is not too different from current practice, such as PVM, NX, Express, p4, etc., and provides extensions that allow greater flexibility.

- Define an interface that can be implemented on many vendor's platforms, with no significant changes in the underlying communication and system software.

- Semantics of the interface should be language independent.

- The interface should be designed to allow for thread-safety.

4 What MPI Does and Does Not Specify

The standard specifies the form of the following.

- Point to point communications, that is, messages between pairs of processes.

- Collective communications: communication or synchronization operations that involve entire groups of processes.

- Process groups: how they are used and manipulated.

- Communicators: a mechanism for providing separate communication scopes for modules or libraries. Each communicator specifies a distinct name space for processes, a distinct communication context for messages and may carry additional, scope-specific information.

- Process topologies: functions that allow the convenient manipulation of process labels, when the processes are regarded as forming a particular topology, such as a Cartesian grid.

- Bindings for Fortran 77 and ANSI C: MPI was designed so that versions of it in both C and Fortran had straightforward syntax. In fact, the detailed form of the interface in these two languages is specified and is part of the standard.

- Profiling interface: the interface is designed so that runtime profiling or performance-monitoring tools can be joined to the message-passing system. It is not necessary to have access to the MPI source to do this and hence, portable profiling systems can be easily constructed.

- Environmental management and inquiry functions: these functions give a portable timer, some system-querying capabilities, and the ability to influence error behavior and error-handling functions.

There are many relevant aspects of parallel programming not covered by the standard. This is also an important list and we give it below.

- shared-memory operations

- interrupt-driven messages, remote execution, and active messages

- program construction tools

- debugging support

- thread support

- process or task management

- input and output functions

The main reason for not addressing these issues was the time constraint self-imposed by the committee, and the feeling that many of them are system dependent. A next set of meetings focused on extending MPI will begin soon.

The remainder of this article discusses some of the more interesting features of MPI.

5 Point to Point Communication

MPI provides a set of send and receive functions that allow the communication of **typed** data with an associated **tag**. Typing of the message contents is necessary for heterogeneous support — the type information is needed so that correct data representation conversions can be performed as data is sent from one architecture to another. The tag allows selectivity of messages at the receiving end: one can receive on a particular tag, or one can wild-card this quantity, allowing reception of messages with any tag. Message selectivity on the source process of the message is also provided.

A fragment of code appears in figure 1 for the example of process 0 sending a message to process 1. This code executes on both process 0 and process 1. The example sends a character string. `MPI_COMM_WORLD` is a default **communicator** provided upon start-up. Among other things, a communicator serves to define the allowed set of processes involved in a communication operation. Process ranks are integers, serve to label processes, and are discovered by inquiry to a communicator (see the call to `MPI_Comm_rank()`). The typing of the communication is evident by the specification of `MPI_CHAR`. The receiving process specified that the incoming data was to be placed in `msg` and that it had a maximum size of 20 elements, of type `MPI_CHAR`. The variable `status`, set by `MPI_Recv()`, gives information on the source and tag of the message and how many elements were actually received. For example, the receiver can examine this variable to find out the actual length of the character string received.

This example employed *blocking* send and receive functions. The send call blocks until the send buffer can be reclaimed (i.e., after the send, process 0 can safely over-write the contents of `msg`). Similarly, the receive function blocks until the receive buffer actually contains the contents of the message. MPI also provides *non-blocking* send and receive functions that allow the possible overlap of message transmittal with computation, or the overlap of multiple message transmittals with one-another. Non-blocking functions always come in two parts: the posting functions, which begin the requested operation; and the test-for-completion functions, which allow the application program to discover whether the requested operation has completed.

This seems like rather a lot to say about a simple transmittal of data from one process to another, but there is even more. To understand why, we examine two aspects of the communication: the semantics of the communication primitives, and the underlying protocols that implement them. Consider the previous example, on process 0, after the blocking send has completed. The question arises: if the send has completed, does this tell us anything about the receiving process? Can we know that the receive has finished, or even, that it has begun?

Such questions of semantics are related to the nature of the underlying protocol implementing the operations. If one wishes to implement a protocol minimizing the copying and buffering of data, the most natural semantics might be the "rendezvous" version, where completion of the send implies the receive has been initiated (at least). On the other hand, a protocol that attempts to block processes for the minimal amount of time will necessarily end up doing more buffering and copying of data.

The trouble is, one choice of semantics is not best for all applications, nor is it best for all architectures. Because the primary goal of MPI is to standardize the operations, yet not sacrifice performance, the decision was made to include all the major choices for point to point semantics in the standard.

An additional, complicating factor is that the amount of space available for buffering is always finite. On some systems the amount of space available for buffering may be small or non-existent. For this reason, MPI does not mandate a minimal amount of buffering, and the standard is very careful about the semantics it requires.

The above complexities are manifested in MPI by the existence of **modes** for point to point communication. Both blocking and non-blocking communications have modes. The mode allows one to choose the semantics of the send operation and, in effect, to influence the underlying protocol of the transfer of data.

In **standard** mode the completion of the send does not necessarily mean that the matching receive has started, and no assumption should be made in the application program about whether the out-going data is buffered by MPI. In **buffered** mode the user can guarantee that a certain amount of buffering space is available. The catch is that the space must be explicitly provided by the application program. In **synchronous** mode a rendezvous semantics between sender and receiver is used. Finally, there is **ready** mode. This allows the user to exploit extra knowledge to simplify the protocol and potentially achieve higher performance. In a ready-mode send, the user asserts that the matching receive already has been posted.

6 User-defined Datatypes

All MPI communication functions take a datatype argument. In the simplest case this will be a primitive type, such as an integer or floating-point number. An important and powerful generalization results by allowing user-defined types wherever the primitive types can occur. These are not "types" as far as the programming language is concerned. They are only "types" in that MPI is made aware of them through the use of type-constructor functions, and they describe the layout, in memory, of sets of primitive types. Through user-defined types, MPI supports the communication of complex data structures such as array sections and structures containing combinations of primitive datatypes. Figure 2 gives an example of using a user-defined type to send the upper-triangular part of a matrix.

7 Collective Communications

Collective communications transmit data among all the processes specified by a communicator object. One function, the barrier, serves to synchronize processes without passing data. Briefly, MPI provides the following collective communication functions.

- barrier synchronization across all processes
- broadcast from one process to all
- gather data from all to one
- scatter data from one to all
- allgather: like a gather, followed by a broadcast of the gather output
- alltoall: like a set of gathers in which each process receives a distinct result
- global reduction operations such as sum, max, min, and user-defined functions
- scan (or prefix) across processes

Figure 3 gives a pictorial representation of broadcast, scatter, gather, allgather, and alltoall. Many of the collective functions also have "vector" variants, whereby different amounts of data can be sent to or received from different processes. For these, the simple picture of figure 3 becomes more complex.

The syntax and semantics of the MPI collective functions was designed to be consistent with point to point communications. However, to keep the number of functions and their argument lists to a reasonable level of complexity, the MPI committee made collective functions more restrictive than the point to point functions, in several ways. One restriction is that, in contrast to point to point communication, the amount of data sent must exactly match the amount of data specified by the receiver. This was done to avoid the need for an array of status variables as an argument to the functions, which would otherwise be necessary for the receiver to discover the amount of data actually received.

A major simplification is that collective functions come in blocking versions only. Though a standing joke at committee meetings concerned the "non-blocking barrier," such functions can be quite useful[1] and may be included in a future version of MPI.

A final simplification of collective functions concerns modes. Collective functions come in only one mode, and this mode may be regarded as analogous to the standard mode of point to point. Specifically, the semantics are as follows. A collective function (on a given process) can return as soon as its participation in the overall communication is complete. As usual, the completion indicates that the caller is now free to access and modify locations in the communication buffer. It does not indicate that other processes have completed, or even started, the operation. Thus, a collective communication may, or may not, have the effect of synchronizing all calling processes. The barrier, of course, is the exception to this statement.

The choice of semantics was done so as to allow a variety of implementations.

The user of MPI must keep these issues in mind. For example, even though a particular implementation of MPI may provide a broadcast with the side-effect of synchronization (the standard allows this), the standard does not *require* this, and hence, any program that relies on the synchronization will be non-portable. On the other hand, a correct and portable program must allow a collective function to be synchronizing. Though one should not rely on synchronization side-effects, one must program so as to allow for it.

Though these issues and statements may seem unusually obscure, they are merely a consequence of the desire of MPI to:

[1]Of course the non-blocking barrier would block at the test-for-completion call.

- allow efficient implementations on a variety of architectures; and,
- be clear about exactly what is, and what is not, guaranteed by the standard.

8 Groups, Contexts, and Communicators

A key feature needed to support the creation of robust, parallel libraries is to guarantee that communication within a library routine does not conflict with communication extraneous to the routine. The concepts encapsulated by an MPI communicator provide this support.

A **communicator** is a data object that specifies the scope of a communication operation, that is, the group of processes involved and the communication context. **Contexts** partition the communication space. A message sent in one context cannot be received in another context. Process ranks are interpreted with respect to the process **group** associated with a communicator. MPI applications begin with a default communicator, `MPI_COMM_WORLD`, which has as process group the entire set of processes (of this parallel job). New communicators are created from existing communicators and the creation of a communicator is a collective operation.

Communicators are especially important for the design of parallel software libraries. Suppose we have a parallel, matrix multiplication routine as a member of a library. We would like to allow distinct subgroups of processes to perform different matrix multiplications concurrently. A communicator provides a convenient mechanism for passing into the library routine the group of processes involved, and within the routine, process ranks will be interpreted relative to this group. The grouping and labeling mechanisms provided by communicators are useful, and communicators will typically be passed into library routines that perform internal communications.

Such library routines can also create their own, unique communicator for internal use. For example, consider an application in which process 0 posts a wildcarded, non-blocking receive just before entry to a library routine. Such "promiscuous" posting of receives is a common technique for increasing performance. Here, if an internal communicator is not created, incorrect behavior could result since the receive may be satisfied by a message sent by process 1 from within the library routine, if process 1 invokes the library ahead of process 0. Another example is one where a process sends a message before entry into a library routine, but the destination process does not post the matching receive until after exiting the library routine. In this case, the message may be received, incorrectly, within the library routine.

These problems are avoided by proper design and usage of parallel libraries. One workable design is for the application program to pass communicators into the library routine that specifies the group and ensures a safe context. Another design has the library create a "hidden" and unique communicator that is set up in a library initialization call, again leading to correct partitioning of the message space between application and library.

Sidebar C shows how one might implement the second type of design. Some thought shows that, as one creates separate communicators for libraries, it is convenient to associate these new communicators with the old communicators from which they were derived. The MPI **caching** mechanism provides a way to set up such an association. Though one can associate arbitrary objects with communicators using caching, the ability to do this for library-internal communicators is one of the most important uses of caching.

9 Conclusions

A pleasant surprise for participants in the MPI effort was the interesting intellectual issues that arose. This article has concentrated on some of these interesting and difficult issues, but for most cases, programming in MPI is straightforward and is similar to programming with other message-passing interfaces.

MPI does not claim to be the definitive answer to all needs. Indeed, our insistence on simplicity and timeliness of the standard precludes that. We believe the MPI interface provides a useful basis for the development of software for message-passing environments. Besides promoting the emergence of parallel software, a message-passing standard provides vendors with a clearly defined, base set of routines that they can implement efficiently. Hardware support for parts of the system is also possible, and this may greatly enhance parallel scalability.

At the final MPI Forum meeting in February 1994, it was decided that plans for extending MPI should wait for more experience with the current version. It seems clear, however, that MPI will soon be expanded in some of the directions listed below.

- Parallel I/O
- Remote store/access
- Active messages
- Process startup
- Dynamic process control
- Non-blocking collective operations
- Fortran 90 and C++ language bindings
- Graphics
- Real-time support

For more information, an MPI-specific newsgroup, `comp.parallel.mpi`, now exists. The official version of the specification document can be obtained from netlib [4] by sending an email message to `netlib@www.netlib.org` with the message: "send mpi-report.ps from mpi". A postscript file will be mailed back to you by the netlib server. The document may also be obtained via anonymous ftp from `www.netlib.org/mpi/mpi-report.ps`, and a hypertext version is available through the world-wide-web at `http://www.mcs.anl.gov/mpi/mpi-report/mpi-report.html`.

References

[1] R. Butler and E. Lusk. Monitors, Messages, and Clusters: The P4 Parallel Programming System. *Parallel Computing*, 20:547–64, April 1994.

[2] R. Calkin, R. Hempel, H. Hoppe, and P. Wypior. Portable Programming with the PARMACS Message–Passing Library. *Parallel Computing, Special issue on message–passing interfaces*, 20:615–32, April 1994.

[3] J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Integrated PVM Framework Supports Heterogeneous Network Computing. *Computers in Physics*, 7(2):166–75, April 1993.

[4] J. Dongarra and E. Grosse. Distribution of Mathematical Software via Electronic Mail. *Communications of the ACM*, 30(5):403–7, July 1987.

[5] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard . *International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4), 1994. Special issue on MPI. Also available electronically, the url is `ftp://www.netlib.org/mpi/mpi-report.ps`.

[6] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994. The book is available electronically, the url is `ftp://www.netlib.org/pvm3/book/pvm-book.ps`.

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

[8] C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr., and M. Zosel. *The High Performance Fortran Handbook.* The MIT Press, 1994.

[9] S. Otto, J. Dongarra, S. Hess-Lederman, M. Snir, and D. Walker. *Message Passing Interface: The Complete Reference.* The MIT Press, 1995.

[10] Parasoft Corporation, Monrovia, CA. *Express User's Guide*, version 3.2.5 edition, 1992. Parasoft can be reached, electronically, at `parasoft@Parasoft.COM`.

[11] 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 767–76. IEEE Press, 1990.

A Sidebar: Implementations of MPI

MPI is available on parallel computers from Convex Computer, Cray Research, IBM, Intel, Meiko, Ncube, NEC, and Silicon Graphics.

A number of public-domain MPI implementations are available and can be found at the following locations.

- Argonne National Laboratory/Mississippi State University implementation. Available by anonymous ftp at `info.mcs.anl.gov/pub/mpi`.
- Edinburgh Parallel Computing Centre CHIMP implementation. Available by anonymous ftp at `ftp.epcc.ed.ac.uk/pub/chimp/release/chimp.tar.Z`.
- Mississippi State University UNIFY implementation. The UNIFY system provides a subset of MPI within the PVM environment, without sacrificing the PVM calls already available. Available by anonymous ftp at `ftp.erc.msstate.edu/unify`.
- Ohio Supercomputer Center LAM implementation. A full MPI standard implementation for LAM, a UNIX cluster computing environment. Available by anonymous ftp at `tbag.osc.edu/pub/lam`.

B Sidebar: More Information on MPI, Assistance

The book by W. Gropp, E. Lusk, and A. Skjellum ([7]) is a tutorial-level explanation of MPI. An expanded and annotated reference manual for MPI is being written by the authors of this article and other members of the MPI Forum, and should be available in 1995.

An MPI-specific newsgroup, `comp.parallel.mpi`, exists. An abundance of information about MPI is available through the world-wide-web. The following is a list of URL's containing MPI-related information.

- Netlib Repository at University of Tennessee and Oak Ridge National Lab (`http://www.netlib.org/mpi/index.html`).
- Argonne National Lab (`http://www.mcs.anl.gov/mpi`).
- Mississippi State University, Engineering Research Center (`http://www.erc.msstate.edu/mpi`).
- Ohio Supercomputer Center, LAM Project (`http://www.osc.edu/lam.html`)
- Australian National University (`file://dcssoft.anu.edu.au/pub/www/dcs/cap/mpi/mpi.html`)

A current version of errata for the specification document ([5]) can be obtained from `ftp://www.netlib.org/mpi/errata.ps`. The complete email associated with the MPI Forum has been archived. They are available from netlib. Send a message to `netlib@ornl.gov` with the message `send index from mpi`. You can also ftp them from `netlib2.cs.utk.edu/mpi`.

So far, at least one company is offering professional support and consulting for MPI. This is PALLAS, and they may be reached at `info@pallas-gmbh.de`.

C Sidebar: Library Communicator and Caching

We wish to give a parallel library its own communicator, with a unique context. The strategy is to pass in, at each invocation of the library, a communicator that describes the process group to be used. The library function "duplicates" it, getting a similar communicator, but one with a unique communication context. This becomes the private, library-internal communicator.

The MPI caching mechanism is used to make this work well. The private communicator is associated (cached) with the communicator passed in by the application. This means that the private communicator needs to be created only the first time the library is invoked with that particular communicator as argument. The caching hides the internal communicator from the application and the application need not explicitly manage the internal communicators. The reader is referred to the further sources discussed in sidebar B for details concerning the caching mechanism.

```
/* static variable used as ``key'' for library */
/* Only one per process is necessary, even if multiple */
/* library invocations can be concurrently active.  */
extern int lib_key;
/* library init.  Need to invoke once by each process, */
/* before library is used.  */
void lib_init()
{
/* allocate a process-unique key */
MPI_Keyval_create(MPI_NULL_FN, MPI_NULL_FN, &lib_key, (void *)NULL);
}
void lib_call( MPI_Comm comm, ...  )
{
int flag;
/* private communicator for library-internal communication */
MPI_Comm *private_comm;
/* retrieve private communicator */
MPI_Attr_get( comm, lib_key, &private_comm, &flag );
if (!flag) {
/* get failed; this is first call and private_comm */
/* has not yet been allocated.  So, do it.  */
/* Make new communicator, with same process group as comm.  */
```

```
private_comm = (MPI_Comm *)malloc(sizeof(MPI_Comm));
MPI_Comm_Dup( comm, private_comm );
/* Cache private communicator with public one.  */
MPI_Attr_put( comm, lib_key, (void *)private_comm );
}
/* Execute library code, using private_comm for */
/* internal communication.  */
...
}
```

```
char msg[20];
int myrank, tag = 99;
MPI_Status status;
...
MPI_Comm_rank( MPI_COMM_WORLD, &myrank ); /* find my rank */
if (myrank == 0) {
strcpy( msg, "Hello there");
MPI_Send( msg, strlen(msg)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
} else {
MPI_Recv( msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status );
}
```

Figure 1: C code. Process 0 sends a message to process 1.

```
double a[100][100];
int disp[100],blocklen[100],i;
MPI_Datatype upper;
...
/* compute start and size of each row */
for (i=0; i<100; ++i) {
disp[i] = 100 * i + i;
blocklen[i] = 100 - i;
}
/* create datatype for upper triangular part */
MPI_Type_indexed( 100, blocklen, disp, MPI_DOUBLE, &upper);
MPI_Type_commit( &upper );
/* ..  and send it */
MPI_Send( a, 1, upper, dest, tag, MPI_COMM_WORLD );
```

Figure 2: A single send transmits the upper-triangular part of a matrix, using a user-defined datatype.

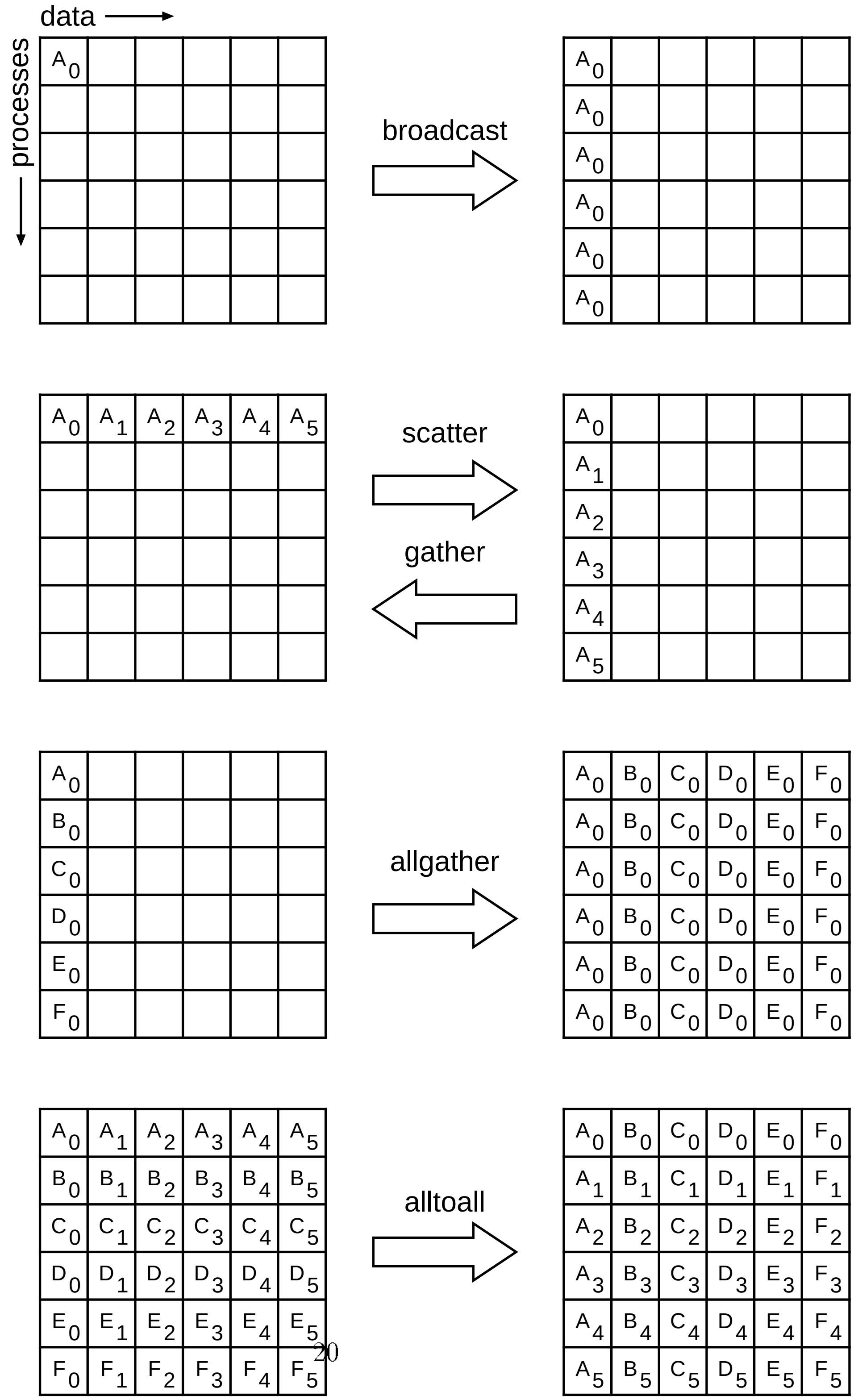

Figure 3: Collective move functions illustrated for a group of six processes. In each case, each row of boxes represents data locations in one process. Thus, in the broadcast, initially just the first process contains the data A_0, but after the broadcast all processes contain it.