The Unify System

Fei-Chen Cheng
Paula Vaughan
Donna Reese
Anthony Skjellum

NSF Engineering Research Center
Mississippi State University

Document for version 0.9.2
September 9, 1994

Contents

Chapter 1

Overview

Unify is a subset of MPI (Message Passing Interface) [4] that has been built on top of PVM (Parallel Virtual Machine). This work is based on a computer science master's project [1] by Fei-Chen Cheng of Mississippi State University, with continued development and production. The initial purpose of this project was to demonstrate that the mappings between MPI and PVM could be implemented in a reasonable time period. The initial version, 0.9.1, represents two months of work.

Unify is a dual-API (Application Program Interface) system. With this system users may write an application containing only MPI calls, or a mixture of MPI and PVM calls [5]. The resulting executable will run in the PVM environment. This subset has been built to show the relative ease of implementation of MPI, and also to ease the migration of code from PVM to MPI. Version 0.9.2 is completely independent of all other MPIs. The other known MPI implementations do not evidently aim at this dual-API migration feature.

Table 1.1 lists a subset of MPI with which a great many parallel applications may be implemented. They are all available wihin this version of *Unify*. The other functions provided by *Unify* are listed in Table 1.2.

1.1 Terminology

The following are some terms with which you should be familiar before continuing in this document.

API Application Program Interface.

Asynchronous Communication With this form of communication, control returns to the calling process before the actual communication action has been completed. This allows the processes perform other processing tasks while waiting for communication to complete. This also requires that the process check for completion of the I/O before reusing the communication buffers.. The burden of insuring uncorrupted data belongs to the process making the call. This is basically the same as nonblocking communication.

Blocking Communication With blocking communication, control is not returned to the calling process until the communication action has been completed. When the function returns, the buffer may be reused. See *Synchronous Communication*.

Function	What it does
MPI_INIT	Initialize MPI
MPI_COMM_SIZE	Find out how many processes there are
MPI_COMM_RANK	Find out which process I am
MPI_SEND	Send a message
MPI_RECV	Receive a message
MPI_FINALIZE	Terminate MPI
MPI_BCAST	One-to-all broadcast
MPI_REDUCE	All-to-one reduction operation
MPI_ALLREDUCE	All-to-all reduction operation
MPI_COMM_DUP	Make a copy of a communicator
MPI_COMM_SPLIT	Partition a communicator into others
MPI_COMM_FREE	Communicator deletion operation
MPI_COMM_GROUP	Group constructor from within communicators
MPI_GROUP_EXCL	Group constructor by deletion of processes
MPI_GROUP_FREE	Group deletion operation

Table 1.1: The six- and fifteen-function versions of MPI.

Collective Communication Collective communication takes place among all processes within a communicator group.

Communicator A communicator is a basic MPI entity. It consists of a group of processes, a communication context and other internal information. Use of a communicator isolates communication within that communicator from other communication taking place within the system.

Context A context is the method by which a communicator ensures the isolation of communication within itself from the rest of the system. It is an internal value, which is not seen by the user.

Group A group is a collection of processes ordered by process rank. Ranks run from 0 to the number of processes minus one.

MPMD Multiple Program/Multiple Data is a form of parallel programming in which more than one program is run in parallel on multiple sets of data.

NonBlocking Communication With nonblocking communication, control returns to the calling process before the communication is complete. See *Asynchronous Communication.*

Point-to-Point Communication Communication between a pair of processes.

SPMD Single Program/Multiple Data is a form of parallel programming in which a single program is run in parallel on multiple pieces of data.

Synchronous Communication In synchronous communication control is not returned to the calling process until the communication action has completed. The process has no requirement to check for completion of the communication and is assured of the

Function	**What it does**
MPI_INITIALIZED	Has MPI_INIT been executed?
MPI_BARRIER	Synchronization function
MPI_GROUP_SIZE	How many processes are in a group?
MPI_GROUP_RANK	What is my rank within the group?
MPI_GET_PROCESSOR_NAME	Get the processor name
MPI_ERROR_STRING	Get error string from error code
MPI_GET_COUNT	Get the length of the message
MPI_ISEND	Asynchronous send
MPI_IRECV	Asynchronous receive
MPI_WAIT	Wait until request is complete
MPI_TEST	See if request is complete
MPI_PROBE	Check for incoming messages
MPI_IPROBE	Asynchronous probe
MPI_SENDRECV	Combined send and receive message
MPI_GATHER	All-to-one function
MPI_SCATTER	One-to-all function
MPI_ALLGATHER	All-to-all gather
MPI_ALLTOALL	Extension of Allgather
MPI_COMM_CREATE	Creation routine for new communicators
MPI_GROUP_TRANSLATE_RANKS	Determine process ranks
MPI_GROUP_UNION	Union two groups
MPI_GROUP_INTERSECTION	Intersect two groups
MPI_GROUP_DIFFERENCE	Find difference between two groups
MPI_GROUP_INCL	Add processes to a group

Table 1.2: Other MPI functions provided with *Unify*.

integrity of the communications buffer when it receives control. This is the same as blocking communication.

1.2 Versions

The current version, 0.9.2, consists of simple point-to-point and collective-communication, communicator-management, and environment-management functions. The C and Fortran bindings for these functions are provided in this version. Internally these MPI calls make use of the PVM features that allow heterogeneity. The initial release supports the SPMD (Single Program/Multiple Data) execution model. Release 1.0 plans to support MPMD (Multiple Program/Multiple Data). The currently implemented functions are described in more detail in chapter 4. A planned schedule of implementation is listed in Table 1.3.

Subsequent versions will increase in functionality until version 1.0 which will be a full implementation of the MPI model, and will integrate with MPIch [2]. We intend full-scale extensions for MPMD computing in version 1.0 as well.

Version	Functionality	Date
V0.9.2	Fortran 77 bindings	(7/94)
V0.9.3	Attribute Caching	(8/94)
V0.9.4	Topologies	(8/94)
V0.9.5	Heterogeneity at the group level	(9/94)
V0.9.6	Inter-communicators	(9/94)
V0.9.7	Dynamic process support / MPMD	(10/94)
V0.9.8	MPIX (portable mode)	(10/94)
V0.9.9	MPIX (native mode)	(11/94)
V1.0	Full implementation of MPI model	(12/94)

Table 1.3: Planned schedule of implementation.

1.3 Changes to PVM

Internally, PVM functions have been modified to add communication context information needed by MPI's protocols. Importantly, *no* changes to the PVM API will be seen from the programmer's perspective. However, not all PVM commands will be useful within the current SPMD programming model currently supported by V0.9.2.

While PVM and MPI functions may be used within the same program, their respective message passing operations do not interact (e.g. An MPI_Recv may not be used to receive a message from a pvm_send).

1.4 How to Get More Information on MPI

The World Wide Web home page for MPI is located at

`http://www.mcs.anl.gov/mpi/index.html`

A version of the MPI standard is on the WWW at

`http://www.mcs.anl.gov/mpi/mpi-report/mpi-report.html`

Information may also be obtained by anonymous ftp from info.mcs.anl.gov in the pub/mpi directory.

A book, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, will be available from the MIT Press in November, 1994 [6].

Another book, *MPI Annotated Reference Manual*, is available from the MIT Press [3].

The newsgroup comp.parallel.mpi will be available in the future.

1.5 Where to Report Problems With *Unify*

Please email any problems with the *Unify* system to the following email address:

`unify-bugs@erc.msstate.edu`

Chapter 2

Installation

This system works with the network version of PVM. It has been tested in the SUN4 (SunOS4.1.x) and SGI environments. Two sets of installation instructions are provided here. Those people who already have PVM and only need to patch it for *Unify* should follow the instructions in the section titled *I Already Have PVM.* If you need a full copy of PVM with *Unify* patches already installed, refer to the section titled *I Don't Have PVM.*

2.1 I Already Have PVM

Two tar files are needed for installation. These are a PVM patches tar file and the *Unify* tar file. At this time patches for PVM versions 3.2, 3.2.6 and 3.3 are supported. The PVM patches are as follows:

```
unipat3.2.0.tar.Z
unipat3.2.6.tar.Z
unipat3.3.0.tar.Z.
```

The *Unify* tar file is

```
unify.tar.Z.
```

Place the appropriate version patch tar file in your $HOME/pvm3 directory. To unpack the *Unify* patch directory type:

```
uncompress unipatx.x.x.tar.Z
tar xvf unipatx.x.x.tar
```

at the Unix prompt.

Now there should be a directory called "unify-patches" on the same level as the patches, src, lib, etc. directories. Go to the unify-patches directory. The unify-patches directory contains the following files:

```
README          lpvmgen.c.diff   pvmumbuf.h.diff
lpvm.c.diff     pvm3.h.diff      unify.patch*
```

If you have the patch command on your system, type

```
unify.patch
```

The patches needed to add MPI to PVM will be performed automatically. If you don't have patch on your system, we recommend that you download the fully patched version of PVM as described in the next section, or get patch (ask your system administrator).

Now that the patched PVM directory is in place, go to your $HOME/pvm3 directory. Type

```
make clean
make all
```

The patched version of PVM must be installed before installing *Unify*. To unpack the *Unify* tar file type:

```
uncompress unify.tar.Z
tar xvfp unify.tar
```

The unify directory contains the following subdirectories:

```
examples/      include/      lib/          src/
```

Change directory to the unify/src directory. Edit the Makefile to point to the appropriate pvm3 and *Unify* directory paths. For example:

```
PVMHOME =      $HOME/pvm3
MPIHOME =      $HOME/unify
```

From within the unify/src directory type

```
make
```

2.2 I Don't Have PVM

Two tar files are needed for installation. These are a PVM tar file and the *Unify* tar file. Fully patched PVM tar files are provided for PVM versions 3.2, 3.2.6, and 3.3. These files are as follows:

```
patpvm3.2.0.tar.Z,
patpvm3.2.6.tar.Z,
patpvm3.3.0.tar.Z.
```

The *Unify* tar file is as follows:

```
unify.tar.Z.
```

If you elect to use the fully patched PVM file, instead of patching PVM yourself, unpack the PVM file by typing the following:

```
uncompress patpvmx.x.x.tar.Z
tar xvf patpvmx.x.x.tar
```

Once the patched PVM directory exists, go to your $HOME/pvm3 directory. Type

```
make clean
make all
```

The patched version of PVM must be installed before installing *Unify*. To unpack the *Unify* tar file type:

```
uncompress unify.tar.Z
tar xvfp unify.tar
```

at the Unix prompt. Change directory to the unify/src directory. Edit the Makefile to point to the appropriate pvm3 and *Unify* directory paths. For example:

```
PVMHOME =      $HOME/pvm3
MPIHOME =      $HOME/unify
```

From within the unify/src directory type

```
make
```

Chapter 3

Programming Environment

3.1 Compiling Unify Applications

A C program that uses *Unify* must be linked to libpvm3.a and libmpi.a. Examples are provided in the unify/examples directory.

3.2 Running Unify Applications

PVM must be running before a *Unify* application is started. Refer to the *PVM 3 User's Guide and Reference Manual* [5] for more information on starting PVM and adding processors to a configuration. While MPI and PVM functions may be used within the same program, the message passing functions of one system will not interact with the other. For example, MPI_Send cannot be used to send a message to a process receiving with pvm_recv.

The number of total processes is a command line argument to a *Unify* executable. To run the program "testprog" with 8 total processes type:

```
testprog -np 8
```

The name of the spawned process is also a command line argument. To run the program "master" and spawn 2 copies of "slave", type

```
master -np 3 -sp slave
```

User command line arguments are passed to the application undisturbed. As with PVM, *Unify* looks for the executables in your $HOME/pvm3/bin/$ARCH directory.

3.2.1 Examples

Examples have been provided in the unify/examples directory. These have been tested on SUN4 (SunOS4.1.x) and SGI platforms.

To prepare the examples for execution go to the unify/examples directory. Edit the Makefile to correct the pvm3 and *Unify* directory paths. For example:

```
PVMHOME =      $HOME/pvm3
MPIHOME =      $HOME/unify
```

Type "make" from within the unify/examples directory. You must now set up symbolic links to the executable from the directory /pvm3/bin/$ARCH. The following example illustrates the establishment of a link to the "first" example program:

```
cd ~/pvm3/bin/\$ARCH
ln -s ~/unify/examples/first .
```

A shell script, "runtest", has been provided to run the test cases within this directory.

3.2.2 Example of a Dual API (MPI and PVM) Program

The following program is a simple example routine in which both MPI and PVM functions have been called. This program divides MPI_COMM_WORLD into three communicators using the MPI_Comm_split function with the *color* option set to *rank* mod 3. Then it passes information in a ring within each of the new communicators using PVM point-to-point communication. The final action is to use the MPI reduction function, MPI_Allreduce, to sum the ranks of the processes within the new communicators.

```
#include "mpi.h"
#include "pvm3.h"

int main( argc, argv )
int argc;
char **argv;
{
    int             rank, size;
    int             gresult;
    int             color, key;
    MPI_Comm        green_comm;
    int             destination, source;
    int             desttid, sourtid;
    int             greensize, greenrank;
    int             pvmdata;

    /* initialize MPI */
    MPI_Init( &argc, &argv );

    /* find out who you are */
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    /* find out how big the process world is */
    MPI_Comm_size( MPI_COMM_WORLD, &size );

    /* prepare to split MPI_COMM_WORLD */
    gresult = 0;

    /* split MPI_COMM_WORLD using rank(mod 3) as       */
    /* the color and using a key which will reverse    */
    /* rank orders                                     */
    color = rank % 3;
    key =  100 - rank;
    MPI_Comm_split( MPI_COMM_WORLD, color, key, &green_comm );
```

```
/* perform some communication with PVM directly   */
/* using the communicator from the previous split */

/* find out the size of the green communicator */
MPI_Comm_size(green_comm, &greensize);

/* find out your rank within the green communicator */
MPI_Comm_rank(green_comm, &greenrank);

/* compute sources and destinations */
source = greenrank - 1;
if (source == -1) source = greensize - 1;
destination = greenrank + 1;
if (destination == greensize) destination = 0;

/* do some PVM point to point communication */

/* find the tid of the destination */
unify_rank_to_tid(green_comm, destination, &desttid);

/* find the tid of the source */
unify_rank_to_tid(green_comm, source, &sourtid);

/* compute some data to exchange */
pvmdata = rank * 30 - 23;

/* send the data */
pvm_initsend(PvmDataDefault);
pvm_pkint(&pvmdata, 1, 1);
pvm_send(desttid, 0);

/* receive the data */
pvm_recv(sourtid, 0);
pvm_upkint(&pvmdata, 1, 1);

/* print the results of the PVM communication */
printf("I am the process with rank %d in\n",greenrank);
printf("the green communicator.\n");
printf("I just received this data, %d,\n",pvmdata);
printf("from rank %d whose tid is %d.\n",source,sourtid);

/* add the ranks of the members of this communicator */
MPI_Allreduce( &rank, &gresult, 1, MPI_INT, MPI_SUM, green_comm);

/* print that result */
printf("The sum of ranks in this communicator is %d.\n",gresult);

/* clean up MPI */
```

```
    MPI_Finalize();
}
```

3.2.3 Example of a Single API (MPI only) Program

This program has the same functionality as the program in section 3.2.2, but it uses only MPI operations.

```
#include "mpi.h"

int main( argc, argv )
int argc;
char **argv;
{
    int                 rank, size;
    int                 gresult;
    int                 color, key;
    MPI_Comm            green_comm;
    int                 destination, source;
    int                 desttid, sourtid;
    int                 greensize, greenrank;
    int                 data;
    MPI_Status          status;

    /* initialize MPI */
    MPI_Init( &argc, &argv );

    /* find out who you are */
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    /* find out how big the process world is */
    MPI_Comm_size( MPI_COMM_WORLD, &size );

    /* prepare to split MPI_COMM_WORLD */
    gresult = 0;

    /* split MPI_COMM_WORLD using rank(mod 3) as       */
    /* the color and using a key which will reverse    */
    /* rank orders                                     */
    color = rank % 3;
    key =  100 - rank;
    MPI_Comm_split( MPI_COMM_WORLD, color, key, &green_comm );

    /* find out the size of the green communicator */
    MPI_Comm_size(green_comm, &greensize);

    /* find out your rank within the green communicator */
    MPI_Comm_rank(green_comm, &greenrank);
```

```
    /* compute sources and destinations */
    source = greenrank - 1;
    if (source == -1) source = greensize - 1;
    destination = greenrank + 1;
    if (destination == greensize) destination = 0;

    /* do some MPI point to point communication */

    /* compute some data to exchange */
    data = rank * 30 - 23;

    /* send the data */
    MPI_Send(&data, 1, MPI_INT, destination, 2002, green_comm);

    /* receive the data */
    MPI_Recv(&data, 1, MPI_INT, source, 2002, green_comm, &status);

    /* print the results of the PVM communication */
    printf("I am the process with rank %d in\n",greenrank);
    printf("the green communicator.\n");
    printf("I just received this data, %d,\n",data);
    printf("from rank %d.\n",source);

    /* add the ranks of the members of this communicator */
    MPI_Allreduce( &rank, &gresult, 1, MPI_INT, MPI_SUM, green_comm);

    /* print that result */
    printf("The sum of ranks in this communicator is %d.\n",gresult);

    /* clean up MPI */
    MPI_Finalize();
}
```

Chapter 4

Function Descriptions

The function descriptions in this chapter are intentionally brief. For more information on these functions and on MPI, refer to the MPI document, *MPI: A Message Passing Interface Standard* [4]. A number of other MPI information sources are listed in chapter 1.4.

4.1 MPI_COMM_WORLD

MPI provides a predefined communicator, MPI_COMM_WORLD. A communicator consists of a group of processes, a communication context and other internal information for use by MPI. Figure 4.1 shows the makeup of a communicator. This special instance of a communicator, MPI_COMM_WORLD, has a group which contains all processes that are accessible after MPI initialization. MPI_COMM_WORLD is depicted in Figure 4.1.

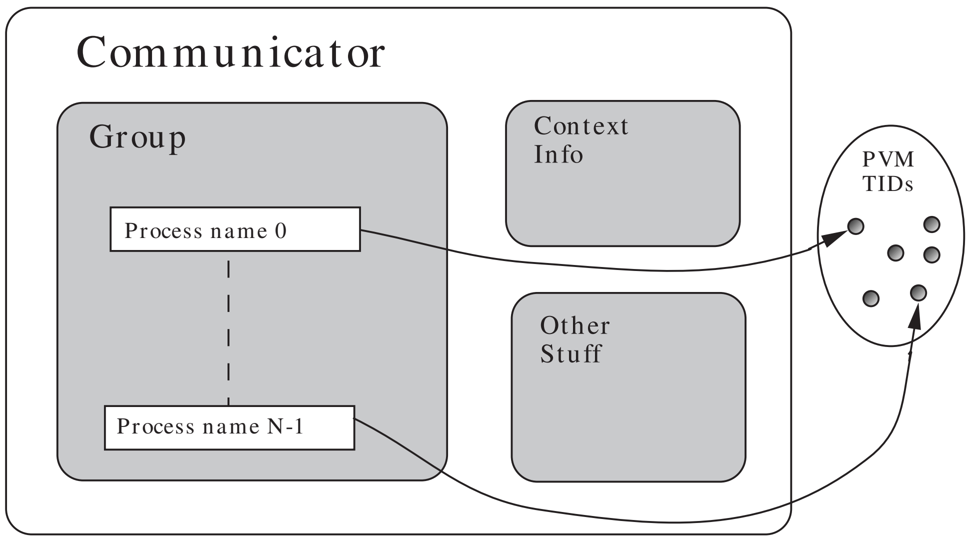

Figure 4.1: A communicator consists of a group, context information, and information internal to MPI.

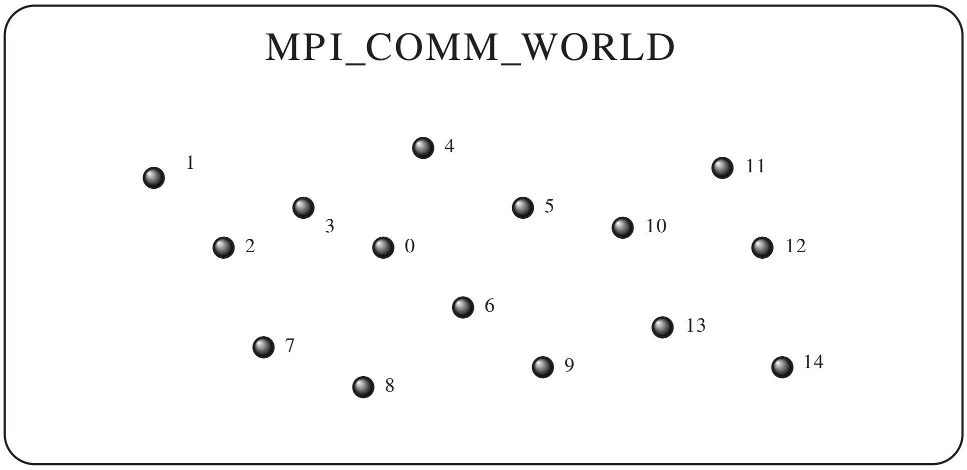

Figure 4.2: MPI_COMM_WORLD. Processes with ranks.

4.2 Other Predefined Quantities

There are other predefined quantities used within *Unify*, which should be understood. Those are listed in Table 4.1.

Quantity	**Meaning**
MPI_COMM_NULL	Invalid communicator
MPI_GROUP_NULL	Invalid Group
MPI_GROUP_EMPTY	Empty Group
MPI_UNDEFINED	Used to specify undefined value
MPI_MAX_PROCESSOR_NAME	Max length of processor name string
MPI_MAX_ERROR_STRING	Max length of error string
MPI_STATUS_SIZE	Size of status array (Fortran)
MPI_REQUEST_NULL	Null communication handle

Table 4.1: Other Predefined Quantities.

4.3 Basic MPI Operations

The fifteen functions represented in this section constitute a fairly reasonable subset of MPI. This subset is sufficient enough to build many useful parallel applications. It may also be valuable in learning the fundamentals of the MPI model. These are the functions that were listed in Table 1.1.

4.3.1 Basic Environmental Functions

MPI_Init

C:

```
int MPI_Init(int  *argc,
```

```
                char ***argv)
```

Fortran:

```
MPI_INIT(IERROR)
    INTEGER IERROR
```

This routine must be called before any other MPI routine. It must be called at most once; subsequent calls are erroneous (*see MPI_Initialized*).

All MPI programs must contain a call to MPI_Init; this routine must be called before any other MPI routine (apart from MPI_Initialized) is called. The version for ANSI C accepts the argc and argv that are provided by the arguments to main:

```
MPI_Init( &argc, &argv );
```

MPI_Finalize

C:

```
int MPI_Finalize(void)
```

Fortran:

```
MPI_FINALIZE(IERROR)
    INTEGER IERROR
```

This routine is called at the end of an MPI program. Its function is to clean up the MPI state. Once this routine is called, no MPI routine (even MPI_init) may be called. The user must ensure that all pending communications involving a process complete before the process calls MPI_Finalize.

4.3.2 Basic Point-to-point Communication

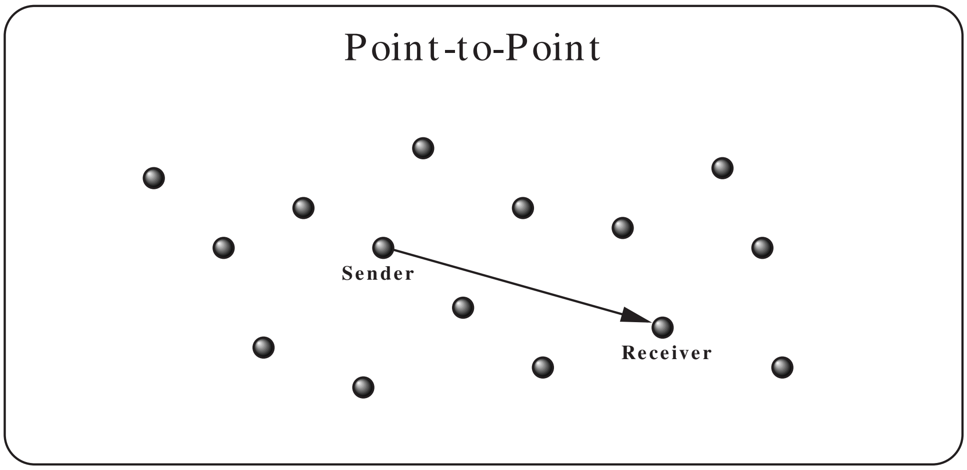

Figure 4.3: Point-to-point communication.

MPI_Send

C:

```
int MPI_Send(void          *buf,
             int          count,
             MPI_Datatype datatype,
             int          dest,
             int          tag,
             MPI_Comm     comm)
```

Fortran:

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

MPI_Send is a blocking send function. This function will not return until it is finished with thethe message data so that the sender is free to access and overwrite the send buffer, *buf*. That means that the blocking send may nontheless return even if no matching receive has been executed by the receiver. Available datatypes are listed in Table 4.2.

MPI_CHAR	character
MPI_SHORT	short integer
MPI_INT	integer
MPI_LONG	long integer
MPI_UNSIGNED_CHAR	unsigned character
MPI_UNSIGNED_SHORT	unsigned short integer
MPI_UNSIGNED	unsigned integer
MPI_UNSIGNED_LONG	unsigned long integer
MPI_FLOAT	floating point
MPI_DOUBLE	double precision floating point
MPI_LONG_DOUBLE	long double precision floating point
MPI_BYTE	byte

Table 4.2: *Unify* supported datatypes.

If the user wishes to use a single name space for processes and a single communication context, the predefined variable MPI_COMM_WORLD may be used as the *comm* argument. This allows communication with all processes defined at initialization time. Point-to-point communication is illustrated in Figure 4.3.2.

MPI_Recv

C:

```
int MPI_Recv(void          *buf,
             int          count,
             MPI_Datatype datatype,
             int          source,
             int          tag,
             MPI_Comm     comm,
             MPI_Status   *status)
```

Fortran:

```
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, Status, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),
            IERROR
```

MPI_Recv is a blocking receive function. The receive buffer consists of the storage containing *count* consecutive elements of the type specified by *datatype*, starting at address *buf*. The length of the received message must be less than or equal to the length of the receive buffer. An overflow error occurs if all incoming data does not fit, without truncation, into the receive buffer.

The *status* variable contains information on the *source* and the *tag* of the received message. They are stored as two fields, MPI_SOURCE and MPI_TAG, in the MPI_Status structure. Point-to-point communication is illustrated in Figure 4.3.2.

4.3.3 Basic Collective Operations

MPI_Bcast

C:

```
int MPI_Bcast(void         *buffer,
              int           count,
              MPI_Datatype  datatype,
              int           root,
              MPI_Comm      comm)
```

Fortran:

```
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
    <type> BUFFER(*)
    INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
```

MPI_Bcast broadcasts a message from the process with rank *root* to all processes of the group, itself included. It is called by all members of group using the same arguments for *comm*, *root*. On return, the contents of root's communication buffer have been copied to all processes. The functionality of MPI_Bcast is depicted in Figure 4.3.3.

MPI_Reduce

C:

```
int MPI_Reduce(void         *sendbuf,
               void         *recvbuf,
               int           count,
               MPI_Datatype  datatype,
               MPI_Op        op,
               int           root,
               MPI_Comm      comm)
```

Fortran:

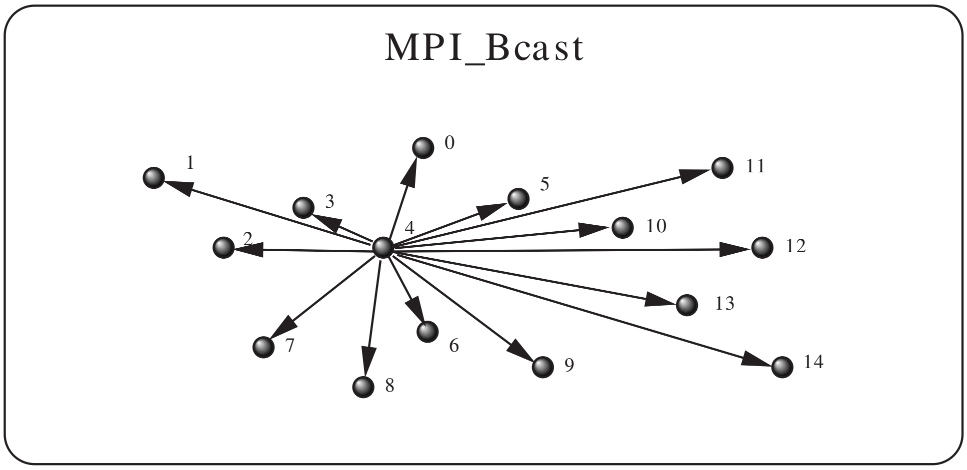

Figure 4.4: MPI_Bcast can be used with any process as the root. This is illustrated in this figure, where the process with rank 4 is the root. With this function, one process can share a piece of information with all other processes within its communicator. *This is a conceptual explanation. It does not represent the implementation.*

```
MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
```

MPI_Reduce combines the elements provided in the input buffer of each process in the group, using the operation *op*, and returns the combined value in the output buffer of the process with rank *root*. The input buffer is defined by the arguments *sendbuf*, *count* and *datatype*; the output buffer is defined by the arguments *recvbuf*, *count* and *datatype*; both have the same number of elements, with the same type. The routine is called by all group members using the same arguments for *count*, *datatype*, *op*, *root* and *comm*. Thus, all processes provide input buffers and output buffers of the same length, with elements of the same type. The functionality of MPI_Reduce is depicted in Figure 4.3.3. Operations supported by the reduce function are listed in Table 4.3.

Operation	Result
MPI_MAX	Maximum value
MPI_MIN	Minimum value
MPI_SUM	Sum of values
MPI_PROD	Product of values
MPI_LAND	Logical AND of values
MPI_BAND	Bitwise AND of values
MPI_LOR	Logical OR of values
MPI_BOR	Bitwise OR of values
MPI_LXOR	Logical XOR of values
MPI_BXOR	Bitwise XOR of values

Table 4.3: Reduction operations.

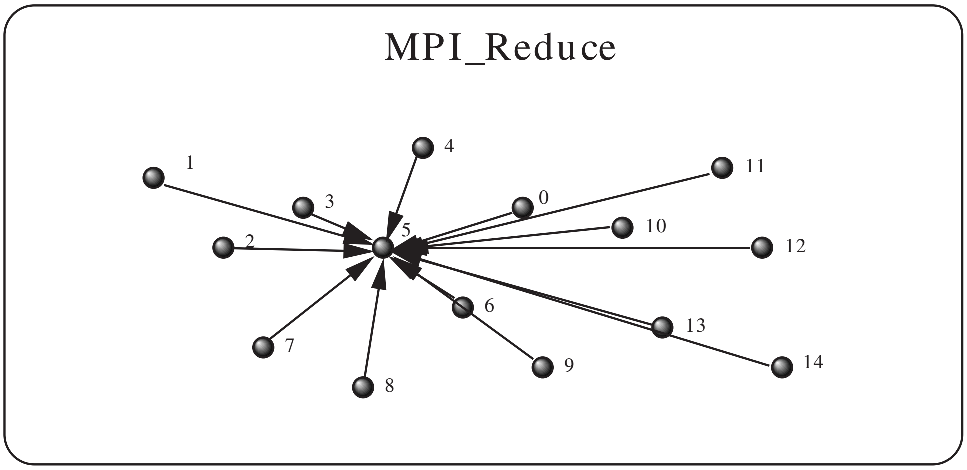

Figure 4.5: MPI_Reduce is an all-to-one function. The information is, in a sense, gathered from all processes. The resulting vector has the specified function performed on it. *This is a conceptual explanation. It does not represent the implementation.*

MPI_Allreduce

C:

```
int MPI_Allreduce(void         *sendbuf,
                  void         *recvbuf,
                  int          count,
                  MPI_Datatype datatype,
                  MPI_Op       op,
                  MPI_Comm     comm)
```

Fortran:

```
MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR
```

MPI_Allreduce is the same as MPI_Reduce except that the result appears in the receive buffer of all the group members. The functionality of MPI_Allreduce is depicted in Figure 4.3.3.

4.3.4 Basic Group, Context, and Communicator Operations

MPI_Comm_group

C:

```
int MPI_Comm_group(MPI_Comm  comm,
                   MPI_Group *group)
```

Fortran:

```
MPI_COMM_GROUP(COMM, GROUP, IERROR)
    INTEGER COMM, GROUP, IERROR
```

MPI_Comm_group returns in group a *handle* to the group of *comm*.

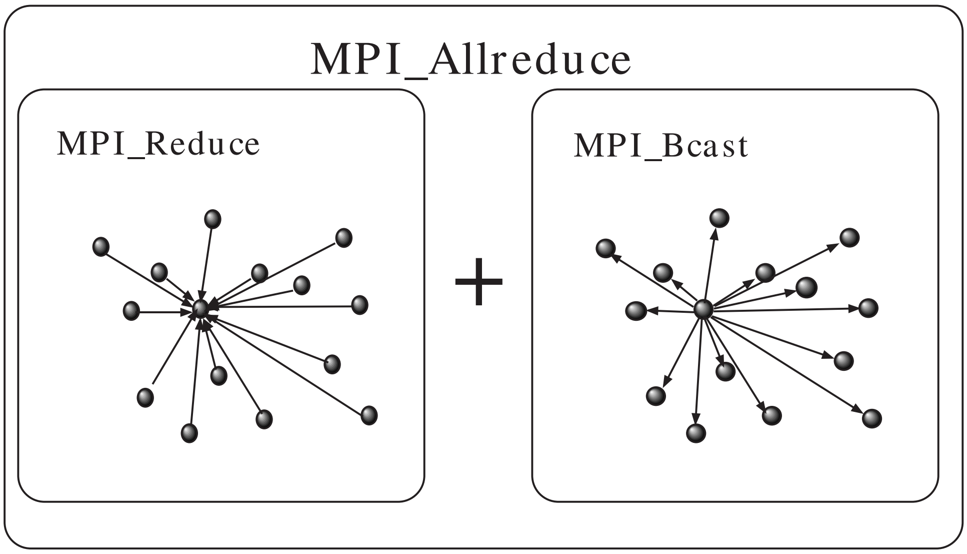

Figure 4.6: MPI_Allreduce is conceptually a reduce followed by a bcast. The data is acted on and reduced to one process, and then broadcasted to the rest of the processes. In the end all processes receive the same information. *This is a conceptual explanation. It does not represent the implementation.*

MPI_Comm_size

C:

```
int MPI_Comm_size(MPI_Comm comm,
                  int      *size)
```

Fortran:

```
MPI_COMM_SIZE(COMM, SIZE, IERROR)
    INTEGER COMM, SIZE, IERROR
```

MPI_Comm_size returns in size the number of processes in the group of communicator comm.

MPI_Comm_rank

C:

```
int MPI_Comm_rank(MPI_Comm comm,
                  int      *rank)
```

Fortran:

```
MPI_COMM_RANK(COMM, RANK, IERROR)
    INTEGER COMM, RANK, IERROR
```

MPI_Comm_rank returns the rank of the calling process within the communicator comm.

MPI_Comm_dup

C:

```
int MPI_Comm_dup(MPI_Comm comm,
                 MPI_Comm *newcomm)
```

Fortran:

```
MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
    INTEGER COMM, NEWCOMM, IERROR
```

MPI_Comm_dup duplicates the existing communicator comm with associated key values. Returns in *newcomm* a new communicator with the same group, any copied cached information, but a new context.

MPI_Comm_split

C:

```
int MPI_Comm_split(MPI_Comm comm,
                   int      color,
                   int      key,
                   MPI_Comm *newcomm)
```

Fortran:

```
MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)
    INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR
```

This function partitions the group associated with *comm* into disjoint subgroups, one for each value of *color*. Each subgroup contains all processes of the same *color*. Within each subgroup, the processes are ranked in the order defined by the value of the argument *key*, with ties broken according to their rank in the old group. A new communicator is created for each subgroup and returned in *newcomm*. A process may supply the *color* value MPI_UNDEFINED, in which case *newcomm* returns MPI_COMM_NULL. This is a collective call, but each process is permitted to provide different values for *color* and *key*.

MPI_Comm_free

C:

```
int MPI_Comm_free(MPI_Comm *comm)
```

Fortran:

```
MPI_COMM_FREE(COMM, IERROR)
    INTEGER COMM, IERROR
```

This collective operation marks the communication object for deallocation. The handle is set to MPI_COMM_NULL. Any pending operations that use this communicator will complete normally; the object is actually deallocated only if there are no other active references to it.

MPI_Group_excl

C:

```
int MPI_Group_excl(MPI_Group group,
                   int       n,
                   int       *ranks,
                   MPI_Group *newgroup)
```

Fortran:

```
MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANKS, NEWGROUP, IERROR
```

MPI_Group_excl creates a group of processes, *newgroup*, that is obtained by deleting from *group* those processes with ranks, *ranks[0]*, ..., *ranks[n-1]*. The ordering of processes in *newgroup* is identical to the ordering in *group*.

MPI_Group_free

C:

```
int MPI_Group_free(MPI_Group *group)
```

Fortran:

```
MPI_GROUP_FREE(GROUP, IERROR)
    INTEGER GROUP, IERROR
```

This operation marks a group object for deallocation. The handle *group* is set to MPI_GROUP_NULL by the call. Any on-going operation using this group will complete normally.

4.4 Other MPI Operations

4.4.1 Other Environmental Operations

MPI_Initialized

C:

```
int MPI_Initialized(int *flag)
```

Fortran:

```
MPI_INITIALIZED(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR
```

This routine may be used to determine whether MPI_Init has been called. It is the only routine that may be called before MPI_Init is called.

MPI_Get_processor_name

C:

```
int MPI_Get_processor_name(char *name,
                           int  *resultlen)
```

Fortran:

```
MPI_GET_PROCESSOR_NAME(NAME, RESULTLEN, IERROR)
    CHARACTER*(*) NAME
    INTEGER RESULTLEN, IERROR
```

This routine returns the name of the processor on which it was called at the moment of the call. The *name* is a character string for maximum flexibility. From this value it must be possible to identify a specific piece of hardware; possible values include "processor 9 in rack 4 of mpp.cs.org" and "231" (where 231 is the actual processor number in the running homogeneous system). The argument *name* must represent storage that is at least MPI_MAX_PROCESSOR_NAME characters long. MPI_Get_processor_name may write up to this many characters into name.

The number of characters actually written is returned in the output argument, *resultlen*.

MPI_Error_string

C:

```
int MPI_Error_string(int  errorcode,
                     char *string,
                     int  *resultlen)
```

Fortran:

```
MPI_ERROR_STRING(ERRORMODE, STRING, RESULTLEN, IERROR)
    CHARACTER*(*) STRING
    INTEGER ERRORMODE, RESULTLEN, IERROR
```

MPI_Error_String returns the error string associated with an error code. The argument *string* must represent storage that is at least MPI_MAX_ERROR_STRING characters long.

4.4.2 Other Point_to_point Communication

MPI_Get_count

C:

```
int MPI_Get_count(MPI_Status   status,
                  MPI_Datatype datatype,
                  int          *count)
```

Fortran:

```
MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
```

The *status* argument of the MPI_Recv function also includes information on the length of the message received. To obtain this information the MPI_Get_count function must be used. *count* is the number of elements of type *datatype* received (not bytes).

MPI_Isend

C:

```
int MPI_Isend(void         *buf,
              int          count,
              MPI_Datatype datatype,
              int          dest,
              int          tag,
              MPI_Comm     comm,
              MPI_Request  *request)
```

Fortran:

```
MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

MPI_Isend is a nonblocking, asynchronous, send function. A nonblocking send call indicates that the sender should not modify any part of the send buffer after the send operation is called, until that send completes. Asynchronous means that the operation does not wait for completion before returning to the user.

MPI_Irecv

C:

```
int MPI_Irecv(void         *buf,
              int          count,
              MPI_Datatype datatype,
              int          source,
              int          tag,
              MPI_Comm     comm,
              MPI_Request  *request)
```

Fortran:

```
MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
    <type> BUF
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
```

MPI_Irecv is the nonblocking, asynchronous, receive function. A nonblocking receive call indicates that the system may start writing data into the receive buffer. The receiver should not access any part of the receive buffer after a nonblocking receive operation is called, until the receive completes.

MPI_Wait

C:

```
int MPI_Wait(MPI_Request *request,
             MPI_Status  *status)
```

Fortran:

```
MPI_WAIT(REQUEST, STATUS, DATATYPE, IERROR)
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR
```

A call to MPI_Wait returns when the operation identified by *request* is complete. If the communication object associated with this request was created by a nonblocking send or receive call, then the object is deallocated by the call to MPI_Wait and the request handle is set to MPI_REQUEST_NULL. MPI_Wait can be used to complete both sends and receives.

MPI_Test

C:

```
int MPI_Test(MPI_Request *request,
             int         *flag,
             MPI_Status  *status)
```

Fortran:

```
MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
    LOGICAL FLAG
```

A call to MPI_Test returns a *flag* value of true if the operation identified by request is complete. In such a case, the *status* object is set to contain information on the completed operation; if the communication object was created by a nonblocking send or receive, then it is deallocated and the request handle is set to MPI_REQUEST_NULL. The call returns *flag* = false, otherwise. In this case, the value of the status object is undefined. MPI_Test can be used to complete both sends and receives.

MPI_Iprobe

C:

```
int MPI_Iprobe(int        source,
               int        tag,
               MPI_Comm   comm,
               MPI_Status *status)
```

Fortran:

```
MPI_IPROBE(SOURCE, TAG, COMM, STATUS, IERROR)
    INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

The MPI_Probe and MPI_Iprobe operations allow incoming messages to be checked for, without actually receiving them. The user can then decide how to receive them, based on the information returned by the probe (basically, the information returned by *status*). In particular, the user may allocate memory for the receive buffer, according to the length of the probed message.

MPI_Probe

C:

```
int MPI_Probe(int        source,
              int        tag,
              MPI_Comm   comm,
              MPI_Status *status)
```

Fortran:

```
MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
    INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

See above.

MPI_Sendrecv

C:

```
int MPI_Sendrecv(void         *sendbuf,
                 int          sendcount,
                 MPI_Datatype sendtype,
                 int          dest,
                 int          sendtag,
                 void         *recvbuf,
                 int          recvcount,
                 MPI_Datatype recvtype,
                 int          source,
                 MPI_Datatype recvtag,
                 MPI_Comm     comm,
                 MPI_Status   *status)
```

Fortran

```
MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG,
             RECVBUF, RECVCOUNT, RECVTYPE, SOURCE, RECVTAG,
             COMM, STATUS, IERROR)
    <type> SENDBUF, RECVBUF
    INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT,
            RECVTYPE, SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE),
            IERROR
```

The send-receive operations combine in one call the sending of a message to one destination and the receiving of another message, from another process. The two (*source* and *destination*) are possibly the same. A send-receive operation is very useful for executing a shift operation across a chain of processes. MPI_Sendrecv is a blocking, synchronous function.

4.4.3 Other Collective Operations

MPI_Barrier

C:

```
int MPI_Barrier(MPI_Comm comm)
```

Fortran:

```
MPI_BARRIER(COMM, IERROR)
    INTEGER COMM, IERROR
```

MPI_Barrier blocks the caller until all group members have called it. The call returns at any process only after all group members have entered the call.

MPI_Gather

C:

```
int MPI_Gather(void         *sendbuf,
               int          sendcount,
               MPI_Datatype sendtype,
               void         *recvbuf,
               int          recvcount,
               MPI_Datatype recvtype,
               int          root,
               MPI_Comm     comm)
```

Fortran:

```
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE,
               RECVBUF, RECVCOUNT, RECVTYPE,
               ROOT, COMM, IERROR)
    <type> SENDBUF, RECVBUF
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
            COMM, IERROR
```

Each process (root process included) sends the contents of its send buffer to the *root* process. The *root* process receives the messages and stores them in rank order. The outcome is as if each of the n processes in the group (including the root process) had executed a call to

```
MPI_Send(sendbuf,sendcount,sendtype,root,...),
```

and the *root* had executed n calls to

```
MPI_Recv(recvbuf+i*recvcount*extent(recvtype),recvcount,i,...),
```

where extent(recvtype) is the type extent obtained from a call to MPI_Type_extent(). The functionality of MPI_Gather is depicted in Figure 4.4.3.

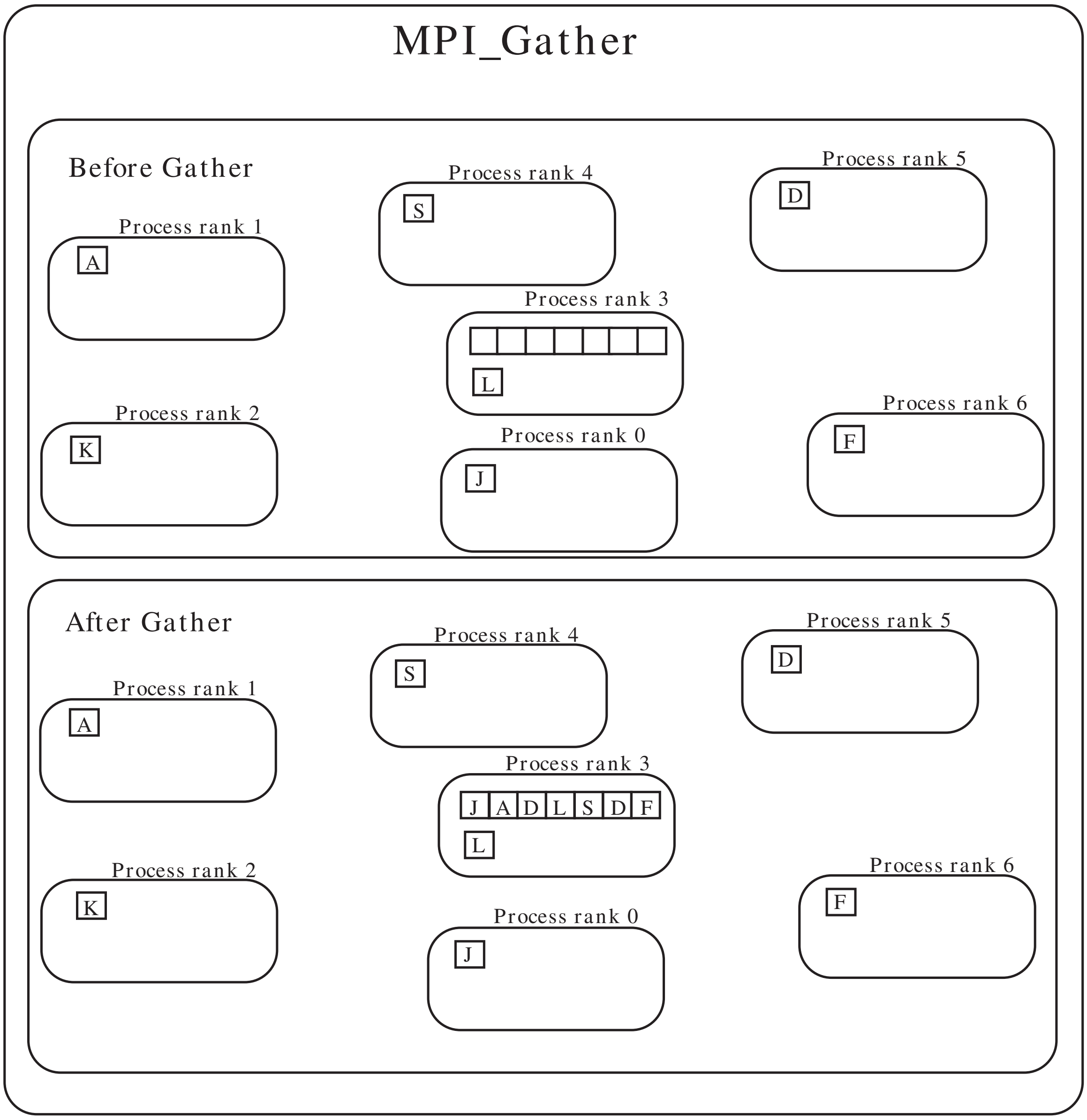

Figure 4.7: MPI_Gather is an all-to-one function. The process specified as root will receive the information sent by each of the other processes. This information will be stored in an array, number of processes long, in process rank order. In this example process with rank 3 has been designated as root. Before the function is performed the destination array has no (or meaningless) information. After the gather function the information from each process has been gathered in the destination array of process rank 3 in rank order.

MPI_Scatter

C:

```
int MPI_Scatter(void         *sendbuf,
                int          sendcount,
                MPI_Datatype sendtype,
                void         *recvbuf,
                int          recvcount,
                MPI_Datatype recvtype,
                int          root,
                MPI_Comm     comm)
```

Fortran:

```
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE,
            RECVBUF, RECVCOUNT, RECVTYPE,
            ROOT, COMM, IERROR)
    <type> SENDBUF, RECVBUF
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE,
            ROOT, COMM, IERROR
```

MPI_Scatter is the inverse operation to MPI_Gather. The outcome is as if the *root* executed n send operations,

```
MPI_Send(sendbuf+i*sendcount*extent(sendtype),
         sendcount,sendtype,i,...),
```

and each process executed a receive,

```
MPI_Recv(recvbuf,recvcount,recvtype,i,...).
```

The functionality of MPI_Scatter is depicted in Figure 4.4.3.

MPI_Allgather

C:

```
int MPI_Allgather(void         *sendbuf,
                  int          sendcount,
                  MPI_Datatype sendtype,
                  void         *recvbuf,
                  int          recvcount,
                  MPI_Datatype recvtype,
                  MPI_Comm     comm)
```

Fortran:

```
MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE,
              RECVBUF, RECVCOUNT, RECVTYPE,
              COMM, IERROR)
    <type> SENDBUF, RECVBUF
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE,
            COMM, IERROR
```

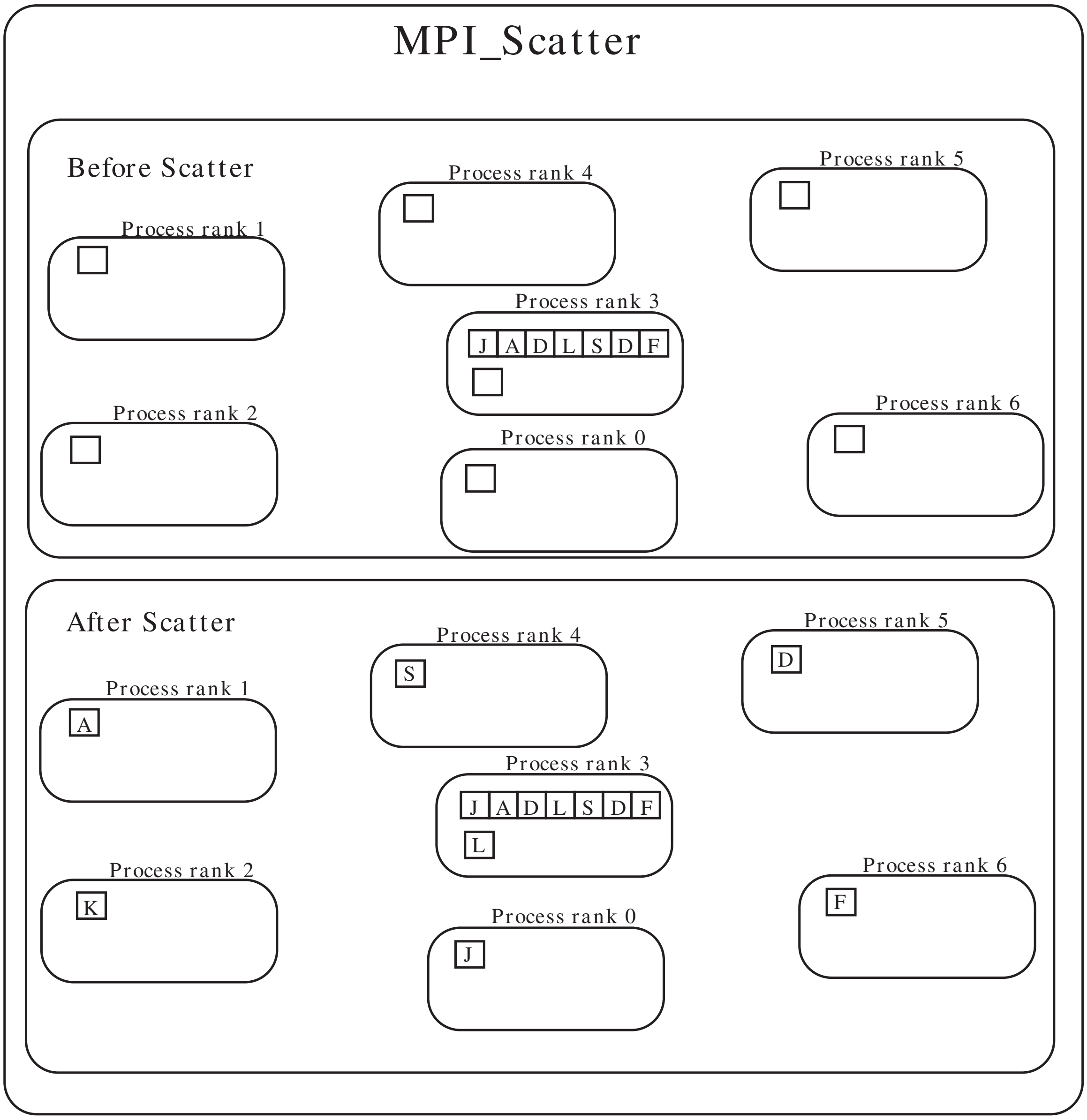

Figure 4.8: MPI_Scatter is a one-to-all function. With this function a process designated with root has a source array which is number of processes long. That information is sent to each of the processes within the root's communicator. Each process receives the information from the array whose position matches its rank. In this example, process with rank 3 is designated as the root. Before the function has been executed, each process's destination has no (or meaningless) value. After the scatter, each process has the information which matched its rank order in the source array.

MPI_Allgather can be thought of as MPI_Gather, but where all processes receive the result, instead of just the root. The jth block of data sent from each process is received by every process and placed in the jth block of the buffer *recvbuf*. The concept of an allgather can be explained as an MPI_Gather followed by an MPI_Bcast.

MPI_Alltoall

C:

```
int MPI_Alltoall(void         *sendbuf,
                 int          sendcount,
                 MPI_Datatype sendtype,
                 void         *recvbuf,
                 int          recvcount,
                 MPI_Datatype recvtype,
                 MPI_Comm     comm)
```

Fortran:

```
MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE,
             RECVBUF, RECVCOUNT, RECVTYPE,
             COMM, IERROR)
    <type> SENDBUF, RECVBUF
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVCOUNT,
            COMM, IERROR
```

MPI_Alltoall is an extension of MPI_Allgather to the case where each process sends distinct data to each of the receivers. The j th block sent from process i is received by process j and is placed in the ith block of *recvbuf*.

4.4.4 Other Group, Context, and Communicator Operations

MPI_Comm_create

C:

```
int MPI_Comm_create(MPI_Comm comm,
                    MPI_Group group,
                    MPI_Comm *newcomm)
```

Fortran:

```
MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
    INTEGER COMM, GROUP, NEWCOMM, IERROR
```

This function creates a new communicator *newcomm* with communication group defined by *group* and a new context. No cached information propagates from *comm* to *newcomm*. The function returns MPI_COMM_NULL to processes that are not in *group*.

MPI_Group_size

C:

```
int MPI_Group_size(MPI_Group group,
                   int       *size)
```

Fortran:

```
MPI_GROUP_SIZE(GROUP, SIZE, IERROR)
    INTEGER GROUP, SIZE, IERROR
```

MPI_Group_size returns the number of processes in *group* in the *size* field.

MPI_Group_rank

C:

```
int MPI_Group_rank(MPI_Group group
                   int       *rank)
```

Fortran:

```
MPI_GROUP_RANK(GROUP, RANK, IERROR)
    INTEGER GROUP, RANK, IERROR
```

MPI_Group_rank returns the rank of the current process within *group* in the *rank* field.

MPI_Group_translate_ranks

C:

```
int MPI_Group_translate_ranks(MPI_Group group1,
                              int       n,
                              int       *ranks1,
                              MPI_Group group2,
                              int       *ranks2)
```

Fortran:

```
MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2, RANKS2, IERROR)
    INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR
```

This function is important for determining the relative numbering of the same processes in two different groups. For instance, if one knows the ranks of certain processes in the group of MPI_COMM_WORLD, one might want to know their ranks in a subset of that group.

MPI_Group_union

C:

```
int MPI_Group_union(MPI_group group1,
                    MPI_group group2,
                    MPI_group *newgroup)
```

Fortran:

```
MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
```

MPI_Group_union returns in *newgroup* a handle to the group including all elements of *group1*, followed by all elements of *group2*.

MPI_Group_intersection

C:

```
int MPI_Group_intersection(MPI_Group group1,
                           MPI_Group group2,
                           MPI_Group *newgroup)
```

Fortran:

```
MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
```

MPI_Group_intersection returns in *newgroup* a handle to the group including all elements of *group1* that are in *group2*, ordered as in *group1*.

MPI_Group_difference

C:

```
int MPI_Group_difference(MPI_Group group1,
                         MPI_Group group2,
                         MPI_Group *newgroup)
```

Fortran:

```
MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
```

MPI_Group_difference returns in *newgroup* a handle to the group including all elements of *group1* that are not in *group2*, ordered as in *group1*.

MPI_Group_incl

C:

```
int MPI_Group_incl(MPI_Group group,
                   int       n,
                   int       *ranks,
                   MPI_Group *newgroup)
```

Fortran:

```
MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR
```

MPI_Group_incl creates a *newgroup* that consists of the *n* processes in *group* with ranks *rank[0], ... , rank[n-1]*.

4.5 Unify-specific Operations

The functions in this section were developed to facilitate the dual-API goal of the *Unify* system.

4.5.1 unify_rank_to_tid

C:

```
int unify_rank_to_tid(MPI_Comm comm,
                      int      rank,
                      int      *tid)
```

Fortran:

```
UNIFY_RANK_TO_TID(COMM, RANK, TID, IERROR)
   INTEGER COMM, TID, RANK, IERROR
```

This operation is used within a dual-API (PVM and MPI) program. PVM requires the process *tid* (task identifier) to be known in order to communicate. This operation will return the *tid* when provided with the communicator and the *rank* of the process in question.

4.5.2 unify_tid_to_rank

C:

```
int unify_tid_to_rank(MPI_Comm comm,
                      int      tid,
                      int      *rank)
```

Fortran:

```
UNIFY_TID_TO_RANK(COMM, TID, RANK, IERROR
   INTEGER COMM, TID, RANK, IERROR
```

This is the inverse operation of unify_rank_to_tid. When given a known PVM *tid*, this operation will return the process's *rank* within the given communicator.

Bibliography

[1] Fei-Chen Cheng. Unifying the MPI and PVM 3 Systems. Technical report, Mississippi State University — Dept. of Computer Science, May 1994. Master's Report. (Available by anonymous ftp: *cs.msstate.edu*, `pub/reports/feicheng.ps`).

[2] Nathan E. Doss, William Gropp, Ewing Lusk, and Anthony Skjellum. An Initial Implementation of MPI. Technical Report MCS-P393-1193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, 1993. in preparation.

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

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

[5] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM 3 User's Guide and Reference Manual. Technical Report TM-12187, Oak Ridge National Laboratory, 1993.

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