5 MPI : The Message Passing Interface

5.1 The message passing paradigm

In general, message passing means a parallel computational model. A computational model is a conceptual view of what types of operations are available to the program [McBryan94], [Andrews91], [GropLusk94]. It does not include the specific syntax of a particular programming language or library, and it is independent of the underlying hardware.[1]

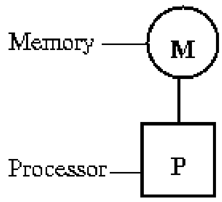

Figure 5-1: The sequential programming paradigm

The sequential paradigm for programming is a familiar one. The programmer has a simplified view of the target machine as a single processor which can access a certain amount of memory (Figure 5-1). He or she therefore writes a single program to run on that processor. The paradigm may in fact be implemented in various ways, perhaps in a time-sharing environment where other processes share the processor and memory, but the programmer wants to remain above such implementation-dependent details, in the sense that the program or the underlying algorithm could in principle be ported to any sequential architecture - that is after all the point of a paradigm.

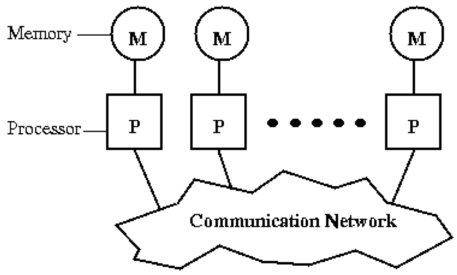

Figure 5-2: The message-passing programming paradigm

The message-passing paradigm is a development of this idea for the purposes of parallel programming. Several instances of the sequential paradigm are considered together. That is, the programmer imagines several processors, each with its own memory space, and writes a program to run on each processor. So far, so good, but parallel programming by definition requires co- operation between the processors to solve a task, which requires some means of communication. The main point of the message-passing paradigm is that the processes communicate by sending each other messages. Thus the message-passing model has no concept of a shared memory space or of processors accessing each other's memory directly. As far as the programs running on the individual processors are concerned, the message passing operations are just subroutine calls. In Figure 5-2 we don't show a specific communication network because it is not part of the computational *model*, rather it is part of the underlying hardware.

In real applications, for the part of the computation local to a process conventional languages like C or Fortran can be used, but subroutine libraries or macros are used to manage the passing of messages between processes.

[1] However, the efficiency of the applications may depend on the gap between the model and the machine.

The message-passing model is not uniformly superior to other parallel computational model (e.g. shared-memory, remote memory operation, etc.), but it has become widely used and it is expected to be around for a long time. Some arguments for message-passing model are listed here:

- **Simplicity.** Sequential languages familiar to most application programmers can be used to perform the bulk of the computation.
- **Universality.** The model fits well on separate processors connected by a (fast or slow) communication network, Thus, it matches the hardware of most today's parallel super-computers, as well as the workstation networks that are beginning to compete with them.
- **Expressivity.** Message-passing has been found to be a useful and complete model in which to express parallel algorithms.
- **Ease of debugging.** Debugging of parallel programs remains a challenging research area. One of the most common causes of error in parallel applications is unexpected overwriting of memory. The message-passing model, by controlling memory references more explicitly than any of other models (only one process has direct access to any memory location), makes easier to locate erroneous memory reads and writes.
- **Performance.** The most compelling reason that message-passing will remain a permanent part of the parallel computing environment is performance. As modern CPUs have become faster, management of their caches and the memory hierarchy in general has become the key to getting the most of them. Message-passing provides a way for the programmer to explicitly associate specific data with processes and thus allow the compiler and cache- management hardware to function fully.

5.2 What is MPI?

MPI is a proposed standard message-passing interface, [MPI Forum94] [GropLusk94]. It is a library specification, not a language. The programs that users can write in Fortran 77 and C are compiled with ordinary compilers and linked with the MPI library. In principle, a sequential algorithm is portable to any architecture supporting the sequential paradigm. However, programmers require more than this: they want their realization of the algorithm in the form of a particular program to be portable - source-code portability. The same is true for message-passing programs and forms the motivation behind MPI. MPI provides source-code portability of message-passing programs written in C or Fortran across a variety of architectures. Just as for the sequential case, this has many benefits, including

- protecting investment in a program
- allowing development of the code on one architecture (e.g. a network of workstations) before running it on the target machine (e.g. fast specialist parallel hardware)

MPI was the first effort to produce a message-passing interface standard across the whole parallel processing community. Sixty people representing forty different organizations - users and vendors of parallel systems from both the US and Europe - collectively formed the "MPI Forum". The discussion was open to the whole community and was led by a working group with in-depth experience of the use and design of message-passing systems (including PVM, PARMACS, p4, etc.). The two-year process of proposals, meetings and review resulted in a document specifying a standard Message Passing Interface (MPI).

Emerging a standard is very important due to the following reasons:

- Portability and ease-of-use.
- Provides hardware vendors with well-defined set of routine to implement efficiently.
- Pre-requisite for the development of concurrent software industry.
- Will lead to more widespread use of concurrent computers.

5.3 General aspects of MPI

5.3.1 What is in MPI ?

MPI was defined to include relatively large number of features that had proven useful in various existing message-passing libraries. According to this, the MPI standard has about 125 functions in it. To keep it manageable, the designers of MPI attempted to make the features of MPI consistent and orthogonal. This means that users can incrementally add sets of functions to their repertoire as needed without learning everything at once. MPI deals with the following areas:

- **Point-to-point message passing.** The simplest form of message is a point to point communication. A message is sent from the sending process to a receiving process. Only these two processes need to know anything about the message. The two basic operations are the send and the receive, but they have a few different versions which represent different semantics in the communication.
- **Collective communication.** A proven concept from existing message-passing libraries is the notion of collective operation, performed by all the processes in a computation, i.e. it allow larger numbers of processes to communicate (e.g broadcast operation). Collective operations are of two kinds:
    1. Data movement operations are used to rearrange data among processes. The simplest of these is a broadcast, but many elaborate scattering and gathering can be defined (and are supported in MPI).
    2. Collective computation operations are used to compute a value from data located different processes, e.g. minimum, maximum, sum, logical OR, etc., as well as user-defined operations.

All of these operations can be built out of point to point communications but it is a good idea use provided routines if they exist. (For example, the message-passing library can take advantage of its knowledge of the structure of the machine to optimize and increase the parallelism in these operations.)

- **Support for process groups.** Processes belong to groups. A process group is an ordered collection of processes, and each process is uniquely identified by its rank within the ordering. For a group of n processes the ranks run from 0 to n - 1. Process groups can be used in two important ways. First, they can be used to specify which processes are involved in a collective communication, such as a broadcast. Second, they can introduce task parallelism into an application, so different groups perform different tasks.
- **Support for communication contexts.** Communication contexts are used to separate families of messages. They promote software modularity by allowing the construction of independent communication streams between processes, thereby ensuring that messages sent in one phase of an application are not incorrectly intercepted by another phase. Communication contexts provides to library writers for the first time the capabilities they need to write parallel libraries that are completely independent of user code and inter-operable with other libraries.
- **Support for 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. Topologies provide a high- level method for managing process groups without dealing with them directly.
- **Profiling interface.** The MPI Forum recognized that profiling and other forms of performance measurement were vital to the success of MPI. At the same time, it seemed far too early to standardize on any particular performance measurement approach. Common to all approaches, however, is the requirement that something particular happens at the time of every MPI call in an application, for example to write a log record. The MPI Forum decided, therefore, to include in MPI a specification for how it would be possible for anyone to intercept calls to the MPI library and perform arbitrary actions.

5.3.2 What is not in MPI ?

Deliberately outside the scope of MPI is any explicit support for:

- Initial loading of processes onto processors
- Spawning of processes during execution
- Multithreading (but MPI is designed to be thread safe)
- Parallel I/O
- Active messages
- Virtual shared memory

5.3.3 Process model and groups

In MPI the fundamental computational unit is the process. The process means the smallest addressable unit of computation. Each process has an independent thread of control, and a separate address space. The MPI process model is static, so as far as the application concerned, a fixed number of processes exist from program initiation to completion. Each process may execute its own distinct code in MIMD style, however MPI does not provide mechanisms for loading executable codes onto processors or assigning processes to processors.[2]

Processes belong to groups. Although MPI process model is static, process groups are dynamic in the sense that they can be created and destroyed, and each process can belong to several groups simultaneously. However, the membership of a group is static : for one or more processes to join or leave a group, a new group must be defined rather then modifying the original one.

Processes are specified - named - by two attributes: a group and a rank relative to the group. For a group of n processes the ranks run from 0 to n - 1.

5.3.4 Separating families of messages

Nearly all message-passing systems provide a tag argument[3] for the send and receive operations. This argument allows the programmer to deal with the arrival of messages in an orderly way, even if the arrival of messages is not in the order desired. The message-passing system queues messages that arrive "of the wrong message tag" until the program(mer) is ready for them. MPI extends the notion of tag with a new concept: the context. Contexts are allocated at run time by the system in response to user requests and are used for matching messages. They partition the "message label space" allowing the construction of independent tag spaces. In MPI, a message label is specified by the message context and the message tag relative to the context.

5.3.5 Communication scope

The "scope" of a communication operation is specified by the context used and the process group involved. The notion of context and group are combined in a single object called a communicator, which becomes an argument to most point-to-point and collective operations. Contexts are not visible at the application level, they are always hidden in the communicator objects. So the programmer must deal with communicators instead of contexts. When a new communicator object is constructed by the programmer the system generate a unique context and put it into the communicator automatically.

5.3.6 Bindings to C and Fortran 77

The MPI standard specifies the format and behaviour of a set of C functions and a similar set of Fortran 77 subroutines. All names of MPI routines and constants in both C and Fortran begin with the prefix MPI_ to avoid name collisions. In the following, we use the C syntax. An MPI function generally has the following format:

```
error = MPI_xxxxx(parameter, ...)
```

[2] Generally the process management is up to the MPI implementations and according to this, it can be significantly different with respect to the different implemented MPI systems.

[3] The tag argument usually is an integer value.

A successful MPI routine will always return the integer constant MPI_SUCCESS, but the be- haviour of an MPI routine which detects an error depends on the error-handler associated with the communicator involved. The default error-handler simple causes the program to abort when an error occurs.Due to this fact, we will ignore the return values of the MPI functions in our example programs.[4] Several implementations of the MPI functions/subroutines exist, covering a wide range of different parallel machines.

5.3.7 General MPI programs

Every MPI program must contain the preprocessor directive

```
#include "mpi"
```

This file, **mpi.h**, contains the definitions, macros and function prototypes necessary for compiling an MPI program. The function **MPI_Init** is used to initialize the MPI stuff, so it must be called before any other MPI function. It must be called at most once, subsequent calls are erroneous. After a program has finished using the MPI library, it must call **MPI_Finalize** in order to clean up all MPI state. Once this routine is called, no MPI routine may be called. So a typical MPI program has the following layout:

```
      ...
#include "mpi.h"
      ...
main(int argc, char** argv) {
      ...
    /* No MPI functions called before this */
    MPI_Init(&argc, &argv);
      ...
    MPI_Finalize();
    /* No MPI functions called after this */
      ...
} /* main */
      ...
```

5.3.8 The first example and the SPMD paradigm

As it is usual in the C-related world, the first example is a version of the classical "Hello, world!" program. The code is as follows:

```
#include <stdio.h>
#include "mpi.h"

int main(argc,argv)
int argc;
char *argv[];
{
    int myrank;      /* Rank of process          */
    int numprocs;    /* Number of processes      */

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);

    printf( "Process %d of %d says Hello!\n",
            myrank, numprocs);

    MPI_Finalize();
}
```

4 However, MPI allows the programmer to define and apply his own error-handler functions instead of the default ones.

The program must be compiled with a C compiler and then must be linked with the MPI library. The way in which MPI programs are "launched" on a particular machine or network is not itself part of the MPI standard. Therefore it may vary from machine to another. Providing that we have the compiled and linked executable called "hello", one would use a command something like:

```
hello -np 4
```

to run this program with four processes (every process executes the same code). The output then should be:

```
Process 1 of 4 says Hello!
Process 2 of 4 says Hello!
Process 0 of 4 says Hello!
Process 3 of 4 says Hello!
```

If the code is run with two processes (e.g. hello -np 2),the output should be:

```
Process 1 of 4 says Hello!
Process 2 of 4 says Hello!
```

Now let us investigate the source code of the program. After defining two integer variables, myrank and numprocs, and initializing the MPI library (MPI_Init()), two MPI function are called : **MPI_Comm_size()** and **MPI_Comm_rank()**. Both of them have two arguments, a communicator and a pointer to an integer variable. They get the default communicator **MP_COMM_WORLD** as them first actual parameters. The **MPI_COMM_WORLD** is a predefined communicator for all MPI application. It defines one context and a process group which contains all processes in the application. The **MPI_Comm_size()** returns (in numprocs) the number of processes that the user has started for this program.[5] The value of numprocs is actually the size of the group associated with the default communicator **MPI_COMM_WORLD**. We think of processes in any group as being numbered with consecutive integers beginning with 0, called *ranks*. Each process finds out its rank in the group associated with a communicator by calling **MPI_Comm_rank()**.

At the end, a printf() is called[6] to make the process print out a hello message together with the values of its myrank and numprocs variables, and then the **MPI_Finalize()** is called which terminates the MPI "environment".

Our hello program uses the *Single Program Multiple Data* or *SPMD* paradigm. That is, we obtain the *effect* of different processes execute different code by taking branches within a single program on the basis of process rank: the statements executed by process 0 are different from those executed by other processes, even though all processes are running the same program. SPMD paradigm is very common approach, especially concerning MPP (Massively Parallel Processing) applications where the number of processes can be very large (even several hundreds).[7]

5.4 Point-to-point communication

A *point-to-point* communication always involves exactly two processes. One process sends a message to the other. This distinguishes it from the other type of communication in MPI, *collective* communication, which involves a whole group of processes at one time.

To send a message, a source process makes an MPI call which specifies a destination process in terms of its rank in the appropriate communicator (e.g. MPI_COMM_WORLD). The destination process also has to make an MPI call if it is to receive the message. Both the source and destination processes must specify the same[8] communicator whose group they must both be in (Figure 5-3).

The basic *send* and *receive* pair of MPI routines have the following C bindings:

---

[5] As MPI has a static process model, the number of processes does not change while the MPI library is used by the application, i.e. "between the **MPI_Init()** and **MPI_Finalize()**".

[6] This sample program is kept as simple as possible by assuming that all processes can do output. However, not all parallel system provide this feature.

[7] The MIMD paradigm is more general than the SPMD one, but essentially they are the same, because it is possible (and easy) to construct the appropriate (equivalent) SPMD version of any MIMD application.

[8] This is due to the fact that every different communicator object has its own unique context.

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

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

In MPI, a message always contains a number of elements of some particular datatype. The first three arguments (that are the same at both functions) define the message buffer in the memory, where the message should come from or where it should be stored. The **(message,count,datatype)** triple describing count occurrences of the data having MPI type **datatype** starting at **buf**. The **datatype** may either be one of the predefined MPI *basic datatypes* or a user-defined **derived datatype** (described later, see Sect. 5.7). The most impor- tant basic MPI types are listed in Table 5-1, together with the corresponding C types.

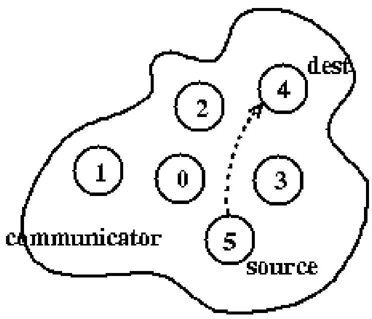

Figure 5-3: Point-to-Point Communication

MPI datatype	**C datatype**
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long_double

Table 5-1: The basic predefined MPI data types

When a message is sent, the receiving process must in general be expecting to receive the same datatype. For example, if a process sends a message with datatype **MPI_INTEGER** the receiving process must specify to receive datatype **MPI_INTEGER**, otherwise the communi- cation is incorrect and behaviour is undefined.

Note that the amount of space allocated for the receiving buffer does not have to match the exact amount of space in the message being received. MPI allows a message to be received as long as there is sufficient storage allocated.

The arguments **dest** and **source** are, respectively, the ranks of the receiving and sending process. MPI allows **source** to be "wildcard". There is a predefined constant **MPI_ANY_SOURCE** that can be used if a process is ready to receive a message from *any* sending process rather than from a particular one. There is *not* a wildcard for dest. Ranks are relatives to the group represented by the communicator **comm** (6-th argument of the functions).

The next argument is the **tag** parameter. It allows the programmer to send some extra information about the actual message, e.g. to distinguish between messages sending in different phase of the process. In the receiving process, the programmer can either apply a particular tag value to receive message only with the particular tag, or can apply the predefined constant **MPI_ANY_TAG** if he is ready to receive message with *any* tag value. MPI guarantees that integers 0 - 32767 can be used as tags, but most implementation allows much larger values. Tags are relatives to the context represented by the communicator comm (6-th argument of the functions).

The communicator **comm** is the next argument, which defines the scope of the communication. The ranks and tags are relatives to the **comm**. There is *no* wildcard for the communicator. In other words, in order for process *A* to send a message to process *B*, the argument **comm** that *A* uses in **MPI_Send** must be identical to the argument that *B* uses in **MPI_Recv**.

The source or tag of a received message may not be known if wildcard values (MPI_ANY_SOURCE, MPI_ANY_TAG) were used in the receive operation. The information is returned by the last ar- gument, status, of **MPI_Recv()**. The **status** is a structure that contains two fields named **MPI_SOURCE** and **MPI_TAG**. Thus, **status.MPI_SOURCE** and **status.MPI_TAG** contain the source and tag, respectively, of the received message. The **status** argument also returns information on the length of the message received. However, this information is not directly available as a field of the **status** variable and a call to **MPI_Get_count** function is required to "decode" this information :

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

It returns (in the **count**) the number of elements received. (It counts *elements*, not *bytes*.) The **datatype** argument should match the argument provided by the receive call that set the **status** variable.

MPI_Send and **MPI_Recv** are both *blocking* communication operation. That is, they return only if the communication operation is locally complete on the process, i.e. if the process has completed its part in the operation. In practice it means the followings:

- **MPI_Send()** does not return until the buffer is "empty" so available for reuse. (The message to be sent has been copied out of the buffer.)
- **MPI_Recv** does not return until the buffer is "full" so available for use. (The message has been arrived and copied into the buffer.)

5.4.1 Example : simple point-to-point communication

We can illustrate the usage of **MPI_Send()** and **MPI_Recv()** by modifying our first example, the "hello" program. Now process with rank 0 prints the greeting messages got from the other processes.

```
#include <stdio.h>
#include "mpi.h"
int main(argc,argv)
int argc;
char *argv[];
{
    int myrank;              /* Rank of process              */
    int numprocs;            /* Number of processes          */
    int source;              /* Rank of sender               */
    int dest;                /* Rank of receiver             */
    char message[100];       /* Storage for the message      */
    MPI_Status status;       /* Return status for receive */

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
```

```
    if (myrank != 0)       /*  My rank is not 0, so I must
                               send greeting */
    { sprintf(message,"Greetings from process
               %d!",myrank);
      dest = 0;
      /* Use strlen(message)+1 to include _\0_ */
      MPI_Send( message, strlen(message)+1,
                MPI_CHAR, dest,15, MPI_COMM_WORLD);
    } else { /* My rank is 0, so I must receive
                the greetings */
      for (source = 1; source < numprocs; source++)
      {  MPI_Recv( message, 100, MPI_CHAR, source,
                   15, MPI_COMM_WORLD, &status);
         printf("%s\n", message);
      }
    }
    MPI_Finalize();
}
```

When the program is compiled and run with four processes, the output should be:

```
Greetings from process 1!
Greetings from process 2!
Greetings from process 3!
```

The C code is rather simple. There is an if statement with two branches. If the process has found that its own rank is not 0, then it executes the first branch. It puts the greeting into the buffer **message**, which is now a character array, and sends it to the process 0. Otherwise, if it is the process with rank 0, then it executes the second branch and receives the greetings from each other process inside a loop. Both send and receive use the default communicator **MPI_COMM_WORLD**. We use the same tag value for sending and receiving (the concrete value is indifferent). However, we might have used the wildcard MPI_ANY_TAG in **MPI_Recv()** as well. Note that process 0 receives the messages, which can have different sizes, in a buffer large enough to store any of them - the buffer has storage for 100 characters.

5.5 Six function MPI

Now we have already known those basic MPI routines which represent the minimal MPI set (see Table 5-2). With only these six functions a vast number of useful and efficient programs can be written.

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

Table 5-2: The six-function version of MPI

The other functions all add flexibility (datatypes), robustness (non-blocking send/receive), efficiency ("ready" mode), modularity (groups, communicators), or convenience (collective operations, topologies). Nonetheless, one can forget all of these concepts and use only the routines from MPI shown in Table 5-2. One can write complete message-passing programs with just these six functions.

5.6 More on point-to-point communication

5.6.1 Blocking and non-blocking communication

The communications described so far are all blocking communications. This means that they do not return until the communication has completed:

- **MPI_Send** does not complete until the buffer can be safely reused (i.e. the message to be sent has copied out of the buffer)
- **MPI_Recv** does not complete until the buffer is ready for use (i.e. the message has received and copied into the buffer)

Communication is not a major user of CPU cycles, but is usually relatively slow because of the communication network and the dependency on the process at the other end of the commu- nication. With blocking communication, the process is waiting idly while the communication is taking place.

MPI provides *non-blocking* communication to allow optimisation by overlapping communication and computation. A non-blocking communication is set up by one command then a later command tests to see if the communication has completed. This allows a DMA engine to copy data while the program is doing other work. So the communication is divided into two oper- ations: the initiation and the completion test. Non-blocking communication is analogous to a form of delegation - the user makes a request to MPI for communication and checks that its request completed satisfactorily only when it needs to know in order to proceed.

The routine **MPI_Isend** begins the nonblocking send operation:

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

The arguments are the same as for **MPI_Send** with the addition of an extra one, called **request**. This argument, **request**, is very important as it provides a *handle* which is used to test when the communication has completed (i.e. when the buffer can be reused). The **MPI_Send** and **MPI_Isend** routines behave similarly except that the latter returns immediately, before the process has completed its part in the communication operation. That is, the buffer containing the message to be sent must not be modified until the completion of the operation is explicitly checked by calling a further appropriate MPI routine.

Similarly, **MPI_Irecv** begins the non-blocking receive operation:

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

It has one additional argument, the handle (**request**) to test the completion, just as **MPI_Isend** does. However it also has one less argument: the status argument, which is used to return information on the completed receive, is deleted from the argument list. The function returns immediately, before the message has arrived and placed into the buffer. The programmer must test with an appropriate MPI routine whether or not the buffer is ready to use.

MPI provides two basic routines to check whether the initiated non-blocking send or receive operation has completed. The first one is the **MPI_Wait**:

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

This routine blocks until the communication specified by the handle **request** has completed. The **request** handle will have been returned by an earlier call to a non-blocking communication routine. If the non-blocking operation is a receive routine then the **status** argument return the information on the completed receive in the same form as **MPI_Recv** does for a blocking receive.

The second basic test routine for non-blocking communication is **MPI_Test**:

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

This routine returns immediately after setting the **flag** variable true, if the operation identified by **request** has completed, or false if it has not. If the communication operation is a receive and the **flag** is set true after **MPI_Test** returning, then the **status** contains just the same information as **MPI_Recv** does.

In many cases, one wishes to test or wait for many non-blocking operations at the same time. MPI provides a way to wait for all or any of a collection of non-blocking operations (with **MPI_Waitall** and **MPI_Waitany**) and to test all or any of a collection of non-blocking operations (with **MPI_Testall** and **MPI_Testany**).

5.6.2 "Unsafe" communication

Although the standard blocking send and receive represent the simpliest way to transfer data between processes, they can be "unsafe". Suppose that we have the following situation, using blocking communication operations:

Process 0	**Process 1**
Send(1)	Send(0)
Recv(1)	Recv(0)

That is, we have two processes which want to communicate with each other. Both of them call the (blocking) send routine first and then the (blocking) receive one.[9] What can happen ? The fundamental problem is that the blocking send operation may not return until the buffers can be safely reused. Hence, if it is impossible to copy the message directly to the buffer defined by the corresponding receive operation - because the corresponding receive operation has not been called yet - then the message must be copied to an internal system buffer in order to allow the process to proceed, or the send operation blocks (until the corresponding receive is called). However, system buffering is outside of the scope of MPI. So the completion of the communications in the above example (Process 0 and Process 1) depends on the details of the actual MPI implementation. There are two possibilities:

1. The MPI implementation have no system buffering at all, or the messages are too large to store in the system buffers, so both send operations are blocked (deadlock situation).
2. There are enough system buffer to store the messages, so the communication operations can be completed successfully.

It is preferable to avoid this kind of "unsafe" communications (i.e. where the completion depends on the system buffering). We list some possible solutions below.

5.6.3 Some solutions to the "unsafe" problem

- **Ordered send and receive.** One of the easiest way to correct for dependence on buffering is to order the sends and receives so that they are paired up. That is, the sends and receives are ordered so that if one process is sending to another, the destination will do a receive that matches the send before doing a send of its own:

Process 0	**Process 1**
Send(1)	Send(0)
Recv(1)	Recv(0)

- **Combined send and receive.** The approach of pairing sends and receives is effective but can be difficult to implement when there are more processes (e.g. shifting data across a chain of processes or across an irregular grid). An alternative is to use the MPI routine **MPI_Sendrecv**:

[9] We use here simplified (not MPI) notations for the blocking routines and only the destination and the source parameters are shown.

Process 0	**Process 1**
Sendrecv(1,1)	Sendrecv(0,0)

The exact C binding for **MPI_Sendrecv** is as follows:

```
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)
```

It executes a blocking send and receive operation in one step. Both send and receive use the same communicator, but possibly different tags. The send buffer and receive buffer must be disjoint, and may have different lengths and datatypes. In our simple example, the actual value of the dest and the source parameters are identical.

- **Buffered sends.** Instead of requiring the programmer to determine a safe ordering of the send and receive operations, MPI allows the programmer to provide a buffer into which data can be placed until it is delivered.

Process 0	**Process 1**
Attach_buffer()	Attach_buffer()
Bsend(1)	Bsend(0)
Recv(1)	Recv(0)

The MPI routine MPI_Bsend uses the buffer provided by the programmer to store the message if the matching receive has not posted yet. Otherwise, MPI_Bsend has ex- actly the same syntax and semantic as MPI_Send does. The programmer can define an appropriate buffer by calling MPI_Buffer_attach :

```
int MPI_Buffer_attach( void* buffer, int size)
```

The buffer is used only by messages sent by calling **MPI_Bsend**[10], and it must be large enough to hold all of the messages that must be sent before the matching receives are called. Only one buffer can be attached to a process at a time.

- Non-blocking operations. Finally, we can avoid the possible deadlock situation by using non-blocking communication operations instead of blocking ones. For example, we can use them in the following arrangement:

Process 0	**Process 1**
Isend(1)	Isend(0)
Irecv(1)	Irecv(0)
Waitall	Waitall

As none of the operations block until the processes reach the Waitall routines, the communications can be completed regardless to the system buffering.

5.6.4 Communication modes

There are four different communication mode for send operation **S**t*andard* (Send), **B***uffered* (**B**send), **S***ynchronous* (**S**send) and **R***eady* (**R**send) (see Table 5-3). So far we use only the standard and the buffered send routines. All four modes exist in both blocking and non-blocking forms. In the blocking forms, return from the routine implies completion. In the non-blocking forms, all modes are tested for completion with the same MPI routines (**MPI_Test**, **MPI_Wait**, etc. see Sect. 5.6.1). One can get the

[10] Or with **MPI_Bsend** which is the non-blocking version of the buffered send and which has the same syntax as **MPI_Isend** does.

name of the non-blocking version of a blocking send routine by simple inserting the letter 'I' at the front of the original name (see Table 5-3).

SEND mode	Blocking	Non-blocking
Standard	**MPI_Send**	**MPI_Isend**
Buffered	**MPI_Bsend**	**MPI_Ibsend**
Synchronous	**MPI_Ssend**	**MPI_Issend**
Ready	**MPI_Rsend**	**MPI_Irsend**

Table 5-3: MPI send routines

The standard, synchronous, ready and buffered sends differ only in one respect: how com- pletion of the send depends on the receipt of the message. All of them have the same argument list. (The arguments are described in Sect. 5.4). We give a short description of the different sends below.

- **Standard send**. (MPI_Send. MPI_Isend) The standard send complete when the buffer can be reused (i.e. the message to be sent is copied out of the buffer). The completion may or may not imply that the message has arrived at its destination. The message may instead lie "in the communications network" (i.e. system buffers) for some time. As system buffering is outside of the cope of MPI, the programmer should assume nothing about the existence or non-existence of such system buffers (see Sect. 5.6.2).
- **Buffered send.** (MPI_Bsend,MPI_Ibsend) Buffered send guarantees to complete immediately, copying the message to the buffer provided by the programmer for later transmission if necessary. (The details of the usage are described in Sect. 5.6.3.) The advantage over standard send is predictability - the sender and receiver are guaranteed not to be synchronised even if there is no system buffering at all. The disadvantage of buffered send is that the programmer must explicitly attach enough buffer space for the program with calls to **MPI_Buffer_attach**. Non-blocking buffered send has no advantage over blocking buffered send.
- **Synchronous send.** (MPI_Ssend, MPI_Issend) Synchronous send does not complete until the matching receive has begun. (Unsafe programs become incorrect and usually deadlock within an MPI_Ssend, see Sect 6.2.) If a process executing a blocking synchronous send is "ahead" of the process executing the matching receive, then it will be idle until the receiving process catches up. Similarly, if the sending process is executing a non- blocking synchronous send, the completion test will not succeed until the receiving process catches up. Synchronous mode can therefore be slower than standard mode, however, it is safer method of communication because the communication network can never become overloaded with undeliverable messages. It has the advantage over standard mode of being more predictable: a synchronous send always synchronises the sender and receiver, whereas a standard send may or may not do so.
- **Ready send**. (MPI_Rsend,MPI_Irsend) A ready send, like buffered send, completes immediately. The communication is guaranteed to succeed normally if a matching receive is already posted. However, unlike all other sends, if no matching receive has been posted, the outcome is undefined (e.g. the message may be silently dropped, or an error occur, etc.). So the programmer must guarantee that matching received has been posted. The idea is that by avoiding the necessity for checking whether the matching receive is already posted and avoiding buffering between the sender and the receiver, performance may be improved.
  Use of ready mode is only safe if the logical control flow of the parallel program permits it. For instance, in Figure 5-4, the synchronisation point - the blocking synchronous send in *Process A* - ensures that the matching receive has already posted when *Process A* calls the ready send function.
  Non-blocking ready send has no advantage over blocking ready send.

While send operations have four different communication mode, there is only one mode for receives. That is, messages sent by different kind of sends all must be received by the *standard* receive routines : **MPI_Recv** or **MPI_Irecv** (see Table 5-4). The receive operations does not complete until the message has arrived and placed into the buffer. The blocking receive (**MPI_Recv** see Sect. 5.4) does not return until the communication operation has completed, while the non-blocking one (**MPI_Irecv** see Sect. 5.6.1) returns immediately and the programmer must test the completion by calling an appropriate MPI function (e.g. **MPI_Test**, **MPI_Wait**, etc.).

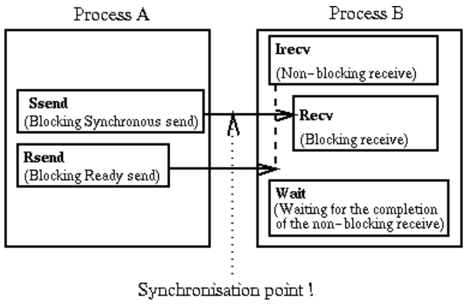

Figure 5-4: An example of safe use of ready mode.

Mode	Blocking	Non-blocking
Standard	**MPI_Recv**	**MPI_Irecv**

Table 5-4: MPI receive routines

5.6.5 Example : circulating in a ring

We ends the discussion about point-to-point communication by a simple example program called "circulation". The aim of the program is to pass a set of numbers around a group of processes. Each process sends its rank to the process rank+1. Each process also receives a rank from the process rank-1. The messages are passed as if the processes are wrapped around, forming a ring, at the ends of the group. The program should terminate once each process receives its own rank value back (i.e. the messages have gone full circle). The source code is as follows:

```
#include "mpi.h"
#include <stdio.h>

int main(argc,argv)
int argc;
char *argv[];
{
   int myid, numprocs, message;
   MPI_Status recv_status, send_status;
   int next, last, in;
   MPI_Request request;
   int tag = 23; /* arbitrary value */

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &myid);
   /* work out identity of neighbours */
   next = (myid + 1) % numprocs;
   last = (myid + numprocs - 1) % numprocs;
   in = -1;
   message = myid;
```

```
   /* messages circle until each process receives its own id back
      again */
   while (in != myid) {
     MPI_Issend( &message, 1, MPI_INT, next, tag,
                 MPI_COMM_WORLD, &request);
     MPI_Recv( &in, 1, MPI_INT, last, tag,
               MPI_COMM_WORLD, &recv_status);
     printf("Process %d received %d\n", myid, in);
     MPI_Wait(&request, &send_status);
     message = in;
   };
   MPI_Finalize();
}
```

When we run the compiled code with three processes the output should be something like this:

```
Process 0 received 2
Process 1 received 0
Process 2 received 1
Process 2 received 0
Process 0 received 1
Process 0 received 0
Process 1 received 2
Process 1 received 1
Process 2 received 2
```

First, the process finds out the number of processes and its own rank in the usual way (**MPI_Comm_size**, **MPI_Comm_rank**). Then it calculates the ranks of its neighbours in the ring and stores these values into the variables `next` and `last`. After that, the circulating of the rank numbers begins inside a `while` loop. In each step, a message is sent to the process with rank next and a message is received from the process with rank `last`. In the first step the own rank of the process is sent while in the further steps the previously received value is sent. The non-blocking synchronous send function (**MPI_Issend**) is used to send the messages. The completion of the send operation is tested by the **MPI_Wait** function (see Sect. 5.6.1) after the blocking receive operation (**MPI_Recv**) has completed. The `while` loop is finished when the process receives its own rank value from the process `last`. Finally, the usual **MPI_Finalize** function is called to terminate the MPI environment.

5.7 Derived data types

In MPI, there is a type argument for all message send and receive operations. In Sect. 5.4 the predefined MPI datatypes have been introduced which correspond to the C base datatypes. As there is a count argument beside the type, we can send and receive arrays of these basic types. However, one might wish to communicate a set of data that is neither an array nor an element of any one MPI basic datatype, for example, coloumns of a matrix which is stored in row-wise manner, or a C struct, or simple a set of general variables. It would be possible to send the data as several messages but this is likely to be inefficient. MPI provides a better solution to this problem, by allowing the user to to specify more general, mixed, and noncontiguous communication buffers at execution time. Before introducing the basic mechanism of creating new MPI datatypes, we must understand how MPI describes a general datatype.

In MPI, a datatype is an object that specifies a sequence of basic datatypes (see Table 5-1) and displacements in bytes, of each of these datatypes. These displacements are taken to be relative to the starting address of the buffer (see Sect. 5.4)that the datatype is describing. A datatype can be presented as a sequence of pairs of basic types and displacements:

Typemap = $\{(type_0; disp_0); ...; (type_{n-1}; disp_{n-1})\}$

MPI calls these sequence the *typemap*. For example, the type **MPI_INT** can be represented as the typemap (int,0). The *type signature* of a datatype is just a list of the basic datatypes in a datatype:

Type signature = $\{type_0; ...; type_{n-1}\}$

The type signature describes what basic types make up an MPI datatype; it is the type signature that controls how data items are interpreted when data is sent or received. In other words, it tell MPI how to interpret the bits in a data buffer. The displacements tell MPI where to find the bits (measured in bytes and relative to the starting address of the buffer). It is important to note, that the type signature of the sending and receiving buffer must be the same in *any* kind of message transfer.

MPI defines some important attributes concerning datatypes, for instance, *size* and *extent*.

The *size* of a datatype is the number of bytes that the data takes up, i.e. the the total size of the data in a message that would be created with this datatype. This is given by the MPI function:

```
int MPI_Type_size(MPI_Datatype datatype, int *size)
```

The first argument is the datatype, and the size is returned in the second argument.

The *extent* of a datatype is defined to be the span from the first byte to the last byte occupied by entries in this datatype, rounded up to satisfy alignment requirements. The data alignment requirements are up to the compiler - in our case the C compiler - so it can be varied. One of the most common requirement is that the address of an item in bytes be multiple of the length of that items in bytes. For example, if an int takes four bytes, then the address of an int must be evenly divisible by four. The extent of a datatype is given by the MPI function:

```
int MPI_Type_extent(MPI_Datatype datatype, int *extent)
```

The first argument is the datatype, the extent is returned in the second one.

To illustrate the difference between the size and extent, consider the typemap: {(*int*; 0); (*char*; 4)} on a computer that requires that int is be aligned on 4-byte boundaries. The size of this datatype is five bytes: 4 + 1 = 5. The first byte is on the location 0, the last byte is on the location 4, so the extent should be 4 - 0 = 4. However, the location of the last byte must be translated due to the alignment requirement : the next int can be placed with displacement eight from the int in the typemap (see Figure 5-5). This makes the extent of this typemap *on the computer we are discussing* eight.

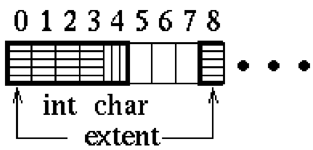

Figure 5-5: Example of datatype extent : the next int can be placed with displacement eight.

The typemap is a completely general way of describing an arbitrary datatype. However, it may not be convenient, particularly if the resulting typemap contains large numbers of entries. MPI provides a number of ways to create new datatypes - derived datatypes - from existing ones, without explicitly constructing the typemap.

Derived datatypes are created at run-time. This is done in two stages:

- **Construct the datatype.** New datatype definitions are built up from existing datatypes (either derived or basic) using a call, or a recursive series of calls to datatype constructors (see below).
- **Commit the datatype.** The new datatype is "committed" with a call to **MPI_Type_commit**. It can then be used in any number of communications. The C binding is:

```
int MPI_Type_commit(MPI_Datatype *datatype)
```

5.7.1 Datatype constructors

In MPI, there are four basic types of datatype constructors:

1. **Contiguous.** The simplest datatype constructor is the **MPI_Type_contiguous** which allows replication of a datatype into contiguous locations. C syntax:

```
int MPI_Type_contiguous( int count, MPI_Datatype oldtype,
                         MPI_Datatype *newtype)
```

newtype is the datatype obtained by concatenating count copies of **oldtype**. Concate- nation is defined using extent as the size of the concatenated copies.

2. **Vector.** The function **MPI_Type_vector** is a more general constructor that allows replication of a datatype into locations that consist of equally spaced blocks. Each block is obtained by concatenating the same number of copies of the old datatype. The spacing between blocks is a multiple of the extent of the old datatype.

```
int MPI_Type_vector( int count, int blocklength, int stride,
                     MPI_Datatype oldtype,
                     MPI_Datatype *newtype)
```

The new datatype **newtype** consists of count blocks each of them consists of **blocklength** copies of **oldtype**. The elements within each block have contiguous displacements, but there is **stride** number of **oldtype** extent space between every block. To construct the particular **newtype** depicted in Figure 5-6, **MPI_Type_vector** function should be called with the following parameters :

- **count**=3
- **blocklength**=2
- **stride**=5

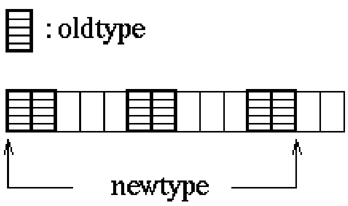

Figure 5-6: Example : Construction of vector *datatype.*

The vector constructer can be used to describe a column of a matrix which is stored in row-wise manner. For example, if we have the matrix data: double data[IMAX][JMAX], we can define an appropriate MPI derived datatype in the following way:

```
MPI_Type_vector( IMAX, 1, JMAX, MPI_DOUBLE,
                 &col_type);
MPI_Type_commit( &col_type)
```

After that, we can send, for example, the second column of the matrix with the following code :

```
MPI_Send(&data[0][1], 1, col_type, ...);
```

3. **Indexed.** The function **MPI_Type_indexed** allows replication of an old datatype into a sequence of blocks (each block is a concatenation of the old datatype), where each block can contain a different number of copies and have a different displacement. All block displacements are multiples of the old type extent.

```
int MPI_Type_indexed( int count,
                      int *array_of_blocklengths,
                      int *array_of_displacements,
                      MPI_Datatype oldtype,
                      MPI_Datatype *newtype)
```

The **newtype** consists of **count** block, where the i-th block is the concatenation of **array_of_blocklengths[i]** number of **oldtype** and has the displacement

array_of_displacements[i]. To construct the particular newtype depicted in Figure 5-7, **MPI_Type_indexed** function should be called with the following parameters :

- **count**=3
- **array_of_blocklengths**=(3,1,4)
- **array_of_displacements**=(0,5,9)

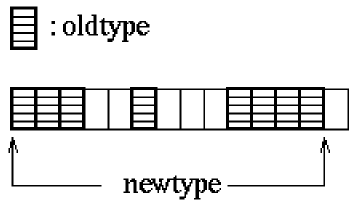

Figure 5-7: Example : Construction of indexed *datatype.*

4. **Struct. MPI_Type_struct** is the most general type constructor. It further generalizes the previous one in that it allows each block to consist of replications of different datatypes. Displacements are given in bytes.

```
int MPI_Type_struct( int count, int *array_of_blocklengths,
                     MPI_Aint *array_of_displacements,
                     MPI_Datatype *array_of_types,
                     MPI_Datatype *newtype)
```

To construct the particular newtype depicted in Figure 5-8, **MPI_Type_struct** function should be called with the following parameters :

- **count**=3
- **array_of_blocklengths**=(1,3,2)
- **array_of_displacements**=(0,10,16)
- **array_of_types**=(MPI_DOUBLE, MPI_CHAR, MPI_INT)

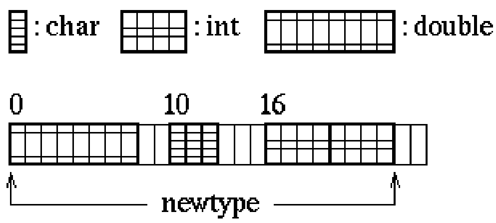

Figure 5-8: Example : Construction of struct *datatype.*

5.8 Collective communication

Collective communication is defined as communication operation which involves a group of processes . An example may be a simple broadcast operation, where one process send the same data to all other processes in a group (see Figure 5-9).

A collective operation is executed by having all processes in the group call the same communication routine, with matching arguments. The message buffer is defined exactly the same way as in the case of point-to-point communication, i.e. with the triple : *(address,count,type)*.

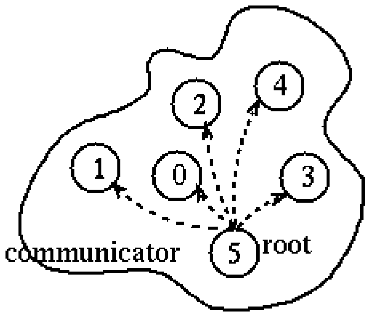

Figure 5-9: Broadcast

One of the key arguments is a communicator that defines the group of participating processes and provides a context for the operation. Collective communication calls may use the same communicators as point-to-point communication; MPI guarantees that messages generated on behalf of collective communication calls will not be confused with messages generated by point- to-point communication.[11]

Several collective routines such as broadcast (see Figure 5-9) have a single originating (or receiving) process. Such processes are called the root. Some arguments in the collective functions are specified as "significant only at root," and are ignored for all participants except the root.

Comparing with point-to-point communication, differences include:

- All collective routine block until the operation has locally completed (i.e. the buffers can safely be used). That is, collective communications have only blocking form, they have no non-blocking versions.
- The amount of data sent must exactly match the amount of data specified by the receiver.
- No message tags are used.

Collective operations can be divided into three main groups: collective data movement, collective computation and synchronisation.

5.8.1 Collective data movement

These routines are provided for rearranging data among processes in a group. Figure 5-10 depicts the schemes of the possible rearrangements.

- **Broadcast** (see Figure 5-10)

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

  MPI_Bcast broadcasts a message from the process with rank root to all processes of the group, itself included. It must called by all members of group using the same arguments for comm, root. The buf, count and datatype arguments are treated as in a point-to-point send on the root and as in a point-to-point receive elsewhere. On return, the contents of root's communication buffer has been copied to all processes.

- **Gather, Scatter** (see Figure 5-10) These routines also specify a root process and all processes must specify the same root and communicator. The main difference from **MPI_Bcast** is that the send and receive details are in general different and so must both be specified in the argument lists. The C binding for the gather operation is:

[11] It is as if each communicator had two contexts, one for point-to-point and one for collective communication.

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

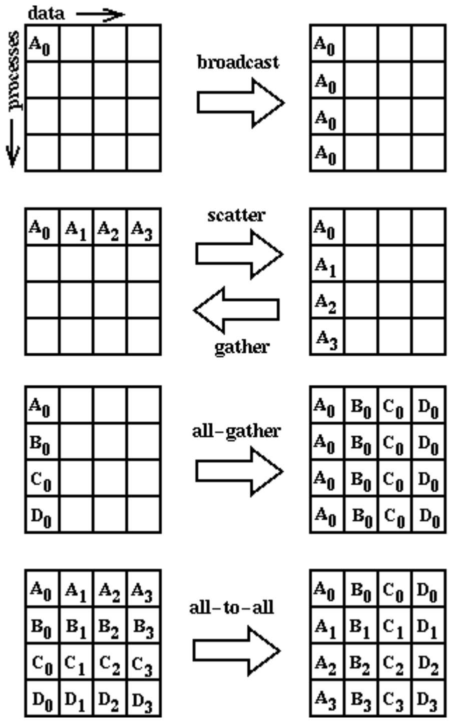

Figure 5-10: Schematic representation of collective data movement in MPI. In each case, each row of boxes represents data locations in one process. Thus, in the broadcast, initially just the first process contains the data A0, but after after the broadcast all process contain it.

Each process in **comm** (**root** process included) sends the contents of its send buffer defined by the triple (**sendbuf**, **sendcount**, **sendtype**) to the process with rank **root**. The process **root** concatenates the received data in process rank order in its receive buffer with starting address **recvbuf**. That is, the data from process 0 is followed by the data from process 1, which is followed by the data from process 2, etc. The argument **recvcount** indicates the number of items received from each process - not the total number received. So the **root** receives **recvcount** number of items from each processes, where the type of each item is **recvtype**. The **recv** arguments are significant only the process with rank root.
Scatter is the "inverse" operation of gather:

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

The process with rank **root** distributes the contents of send buffer with starting address **sendbuf** among the processes in the group defined by the communicator **comm**. The contents of the send buffer are split into segments each consisting of **sendcount** items of type **sendtype**. The first segment goes to process 0, the second to process 1, etc. The send arguments are significant only on process **root**.

- **All-gather** (see Figure 5-10)

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

MPI_Allgather can be thought of as **MPI_Gather**, but where all processes receive the result, instead of just the root. That is, there is no root argument, and the **recv** arguments are significant on every process in the group. Otherwise, the arguments have exactly the same meanings as in the case of **MPI_Gather**.

- **All-to-all** (see Figure 5-10)

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

MPI_Alltoall is an extension of **MPI_Allgather** to the case where each process sends distinct data to each of the receivers. The send buffer with starting address **sendbuf** are split into segments each consisting of **sendcount** items of type **sendtype**. Each process send the first segment of its send buffer to the process 0 which concatenates the received segments in rank order in the receive buffer with starting address **recvbuf**. The second segment of the send buffer is sent to the process 1 by each process, etc.

5.8.2 Collective computation

Collective computation operations are used to compute a value from data located different processes. Typical computation can be global sum, minimum, maximum, etc. Figure 5-11 depicts a schematic representation of the available collective computation patterns.

- **Reduce.** In a collective reduce operation, all the processes (in a group represented by a communicator) contribute data which is combined using a binary operation. The C binding is:

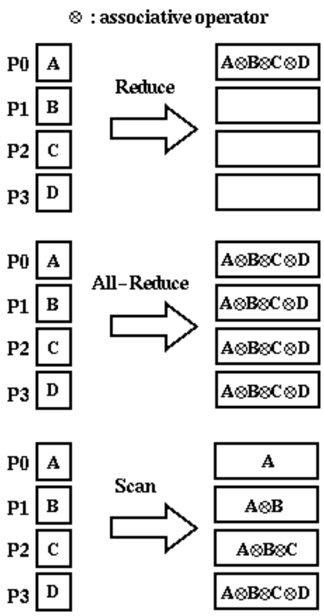

Figure 5-11: Collective Computation Patterns

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

MPI_Reduce combines the operands stored in the operand buffer using operation op and stores the result in the result buffer on the process **root**. The operand buffer is defined by the arguments **operand**, **count** and **datatype**; the result buffer is defined by the arguments **result**, **count** and **datatype**; both have the same number of elements, with the same type. **MPI_Reduce** must be called by all processes in the communicator **comm**, and **count**, **datatype**, and **op** must be the same on each process. The **result** argument is significant only on the root process.
The argument **op** has the type **MPI_Op**, and it can take on one of the predefined values listed in Table 5-5
. It is also possible to define additional operators. It is discussed later.

Operation Name	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical And
MPI_LOR	Logical Or
MPI_LXOR	Logical Exclusive Or
MPI_BAND	Bitwise And
MPI_BOR	Bitwise Or
MPI_BXOR	Bitwise Exclusive Or
MPI_MAXLOC	Maximum and Location of Maximum
MPI_MINLOC	Minimum and Location of Minimum

Table 5-5: Predefined operators for collective computation.

- **All-Reduce.** Same as **MPI_Reduce** except that the result appears in the result buffer of all the group members (see Figure 5-11).

```
int MPI_Allreduce( void* operand, void* result,
                   int count,
                   MPI_Datatype datatype,
                   MPI_Op op, MPI_Comm comm)
```

  Hence the root argument is omitted and the result buffer is significant at each process in the group.

- **Scan. MPI_Scan** performs "parallel prefix" type operations (see Figure 5-11).

```
int MPI_Scan( void* operand, void* result,
              int count, MPI_Datatype datatype,
              MPI_Op op, MPI_Comm comm )
```

  If process *i* holds the single value *v(i)* (in the operand buffer), the result on process *i* is *v(1)* op *v(2)* op... op *v(i)* which is stored in the result buffer. The arguments are as for **MPI_Allreduce**.

It is possible for the user to supply an arbitrary associative binary operator for the collective computation routines. The MPI function:

```
int MPI_Op_create( MPI_User_function *function,
                   int commute, MPI_Op *op)
```

binds the user-defined operation **function** to an **op** handle that can subsequently be used in **MPI_Reduce**, **MPI_Allreduce**, and **MPI_Scan**. The user-defined operation is assumed to be associative. If **commute**=true, then the operation should be both commutative and associative. The type **MPI_User_function** is defined as follows:

```
typedef void MPI_User_function( void *invec,
                                void *inoutvec,
                                int *len,
                                MPI_Datatype *datatype);
```

That is, the user defined function must have four arguments with types (void *, void *, int *, MPI_Datatype *). The function is expected to perform the following computation :

```
for (i = 1 to len)
    inoutvec(i) = inoutvec(i) op invec(i)
```

So ***invec*** and ***inoutvec*** contain the two input operands, and the result must be stored in inoutvec when the function returns. The ***datatype*** argument is point to the MPI data type which was passed to the collective computation routine.

5.8.3 Synchronisation

The simplest routine of all the collective operations is MPI_Barrier:

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

It involves no data at all, and simple blocks until all other members of the group (represented by **comm**) have called it.

It can be very useful in such a situation, for example, when in one phase of a computation, all processes participate in writing a file, but the file is to be used as input data for the next phase of the computation. Therefore no process should proceed to the second phase until all processes have completed phase one.

5.8.4 Example : calculation of the π

To illustrate the usage of collective data movement and collective computation operations let have a look at the following example program, which computes the value of ss by numerical integration. Since

$$\int_0^1 \frac{1}{1+x^2}\,dx = \arctan(x)\Big|_0^1 = \arctan(1) - \arctan(0) = \arctan(1) = \frac{\pi}{4},$$

the integral of the $F(x) = 4 / (1 + x^2)$ function in the [0; 1] interval is equal with π. To do this integration numerically, the program divides the interval from 0 to 1 into some number n of subintervals and approximate the integral in each subinterval with area of a rectangle. Larger values of the parameter n will give us more accurate approximations of π. The parallel part of the algorithm occurs as each process computes and adds up the areas for different subset of rectangles. The C source code is as follows:

```
#include "mpi.h"
#include <math.h>

int main(argc,argv)
int argc;
char *argv[];
{
    int n, myid, numprocs, I;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;

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

```
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    n = 0;
    if (myid == 0)
      { printf("Enter the number of intervals:");
        scanf("%d",&n);
      }

    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    h = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs)
      { x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
      }
    mypi = h * sum;

    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE,
               MPI_SUM, 0, MPI_COMM_WORLD);

    if (myid == 0)
      { printf( "pi is approximately %.16f, Error is
                %.16f\n",
                pi, fabs(pi - PI25DT));
      }
    MPI_Finalize();
}
```

First, each process call the usual three MPI functions to initiate the MPI library and to get the size of the group and its own rank in the predefined communicator **MPI_COMM_WORLD** (see Sect. 5.3.8). Then the process with rank 0 obtains the value of *n* from the user. To transfer this value to every process (i.e. to every process in the **MPI_COMM_WORLD** communicator), all processes perform a broadcast operation (**MPI_Bcast**) with the appropriate arguments. Each process is able to compute which rectangles it is responsible for from *n*, the total number of processes and its own rank. They perform the necessary local computations inside a for loop. The local subresult is stored in the variable `mypi`. At the end, the MPI reduce function is called by all of the processes to compute the total sum from the distributed subresults. The **MPI_Reduce** store the result into the variable `pi` on process 0. Process 0 prints the result and the error in the approximation to the standard output. Finally, every process terminates the MPI environment by calling **MPI_Finalize**.

5.9 Convenient process naming : virtual topologies

A virtual topology is a mechanism for naming the processes in a communicator in a way that fits the communication pattern better. The main aim of this is to makes subsequent code simpler. It may also allow MPI to optimise communications (e.g. to optimise the mapping of the processes to physical processors), however it is up to the MPI implementations.

For example, if our processes will communicate mainly with nearest neighbours after the fashion of a two-dimensional grid (see Figure 5-12), we could create a virtual topology to reflect this fact. In the figure, lines denote the main communication patterns, namely between neighbours. The numbers represent the conceptual coordinates in the grid. This grid actually has a cyclic boundary condition in one direction e.g. processes (0,0) and (0,3) are "connected". In MPI terms, the dimension corresponding the rows of the grid is periodic.

What this gains us is access to convenient routines which, for instance, compute the rank of any process given its coordinates in the grid or the ranks of the nearest neighbours of an arbitrary process in the grid. The rank may then be used as an argument to the diverse send and receive MPI routines.

Although a virtual topology highlights the main communication patterns in a communicator by a "connection", any process within the communicator can still communicate with any other.

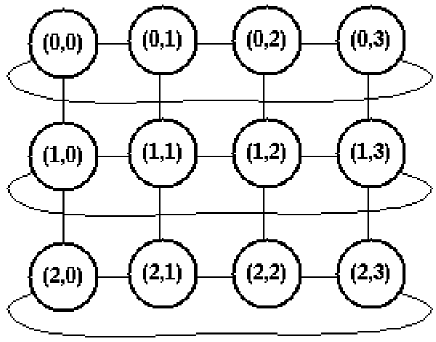

Figure 5-12: Example: 2-D Cartesian Virtual Topology

In MPI, a virtual topology is associated with a communicator. When a virtual topology is created on an existing communicator, a new communicator is automatically created and returned to the user. The user must use the new communicator rather than the old to use the virtual topology.

The user can configure the processes into one of two topology types:

- **Cartesian topologies**. Creating a Cartesian topology "connects" each process to its neighbours in a virtual grid of arbitrarily many dimensions. Each dimension can be specified independently as either periodic (wrap-around) or non-periodic. Functions are provided to allow processes to be identified by Cartesian coordinates.
- **General Graph topologies**. MPI allows completely general graph virtual topologies, in which a process may be "connected" to any number of other processes and the numbering is arbitrary. Each process can retrieve the list of its neighbours in the graph and will normally communicate only with those processes.

We will only describe the usage of Cartesian virtual topologies. The general graph topologies are used in similar way, although of course there is no concept of coordinates ([MPI Forum94]).

5.9.1 Creating Cartesian virtual topologies

```
int MPI_Cart_create( MPI_Comm comm_old,
                     int ndims, int *dims,
                     int *periods, int reorder,
                     MPI_Comm *comm_cart)
```

MPI_Cart_create takes an existing communicator **comm_old** and returns a new communi- cator **comm_cart** with the virtual topology associated with it. The user must use this new communicator in the subsequent MPI function calls when he wants to access the virtual grid. The Cartesian grid can be of any dimension and may be periodic or not in any dimension, so tori, rings, three-dimensional grids, etc. are all supported. The **ndims** argument contains the number of dimensions. The number of processes in each dimension is specified in the array **dims** and the array **periods** is an array of TRUE or FALSE values specifying whether that dimension has cyclic boundaries or not. The **reorder** argument is an interesting one. It can be TRUE or FALSE:

- FALSE. In this case the process ranks in the new communicator (**comm_cart**) remain exactly as in **old_comm**.
- TRUE. It allows the MPI to renumber the process ranks in the new communicator (possibly so as to choose a good embedding of the virtual topology onto the physical machine).

The **reorder** parameter must be set FALSE if the data is already distributed to the processes, otherwise, the TRUE value is preferable because it might increase the run-time performance.

MPI_Cart_create must be called by all processes in the communicator **comm_old** with the same parameters.If the total size of the Cartesian grid is smaller than the size of the group of **comm_old**, then some processes are returned **MPI_COMM_NULL** instead of the new communicator, which implies that they do not participate in the new communicator (i.e. they are not members of the grid). The call is erroneous if it specifies a grid that is larger than the group size.

For example, to get the virtual grid depicted in Figure 5-12, the **MPI_Cart_create** function must be called with the following parameters:

- **ndims**=2
- **dims**=(4,3)
- **periods**=(1,0)

5.9.2 Identifying processes in Cartesian grid

After creating the new communicator which contains the appropriate topology informations, each process may find out its own Cartesian coordinates by calling the MPI function **MPI_Cart_coords**, which converts process rank[12] to Cartesian coordinates:

```
int MPI_Cart_coords( MPI_Comm comm, int rank,
                     int maxdims, int *coords)
```

The function returns the Cartesian coordinates belonging to the process **rank** in communicator **comm**, where **comm** must come from a previous call of **MPI_Cart_create**. The **maxdims** argument specifies the length of the array **coords** (i.e. it is usually equal with the **ndims** parameter used to create the grid).

The inverse function MPI_Cart_rank converts process grid coordinates to process rank.

```
int MPI_Cart_rank( MPI_Comm comm, int *coords, int *rank)
```

It might be used to determine the rank of a particular process whose grid coordinates are known, in order to send a message to it or receive a message from it.

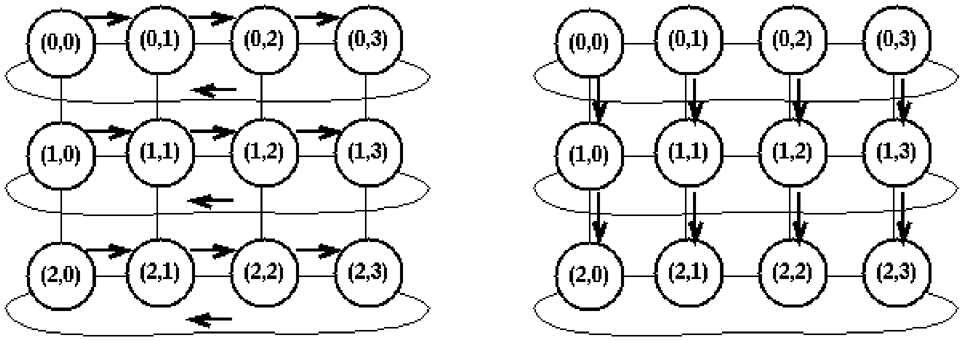

Figure 5-13: Shift operations in a 2-D Cartesian grid.

As **MPI_Cart_rank** represents a general way to determine the rank of an arbitrary member of the grid, MPI provides a more convenient routine to determine the rank of the sender and receiver processes concerning a *shift* operation. Shift is a common operation, which involves a set of processes passing data to each other in a chain-like fashion (or a circular fashion). Figure 5-13 depicts two shift operations in a grid, the first one takes place in the first dimension, (which is a periodic one), while the second shift is performed in the second dimension (which is a non-periodic one). The function :

```
int MPI_Cart_shift( MPI_Comm comm,
                    int direction, int disp,
                    int *rank_source, int *rank_dest)
```

does not actually What it does do is return the correct ranks for a shift which can then be included directly as arguments to MPI send and receive routines[13], to perform the shift. Suppose every process in a grid ($x; y$) is to send a piece of data to its neighbour ($x; y + 1$). It will also receive a piece of data

[12] To find out its rank in the new communicator, a process may use the usual **MPI_Comm_rank** function (see Sect. 5.3.8).

[13] The **MPI_Sendrecv** function (see Sect. 5.6.3) is to be the most simple and natural way to perform shift operation in a grid.

from ($x; y$ - 1). Each process can find the ranks of the processes it has to send data to and receive data from by calling **MPI_Cart_shift** with **direction**=1 (specifying the dimension in which the shift is to occur) and **disp**=1 (specifying the distance of the shift).[14] If the specified dimension is periodic, all processes get valid ranks for **rank_source** and **rank_dest** by wrapping around at the ends. However, for non-periodic dimensions, the special value **MPI_PROC_NULL** is returned if the corresponding co-ordinates specify a point outside the valid process range. Any send or receive specifying **MPI_PROC_NULL** as the rank of source or target process will return immediately successfully without attempting any communication.

5.9.3 Cartesian partitioning

Quite often, a program with a Cartesian topology may need to perform reduction operations or other collective communications only on rows or columns of the grid rather than the whole grid. Figure 5-14 depicts two simple examples. **MPI_Cart_sub** exists to create new communicators for sub-grids or "slices" of a grid.

```
int MPI_Cart_sub(MPI_Comm comm,
                 int *remain_dims, MPI_Comm *newcomm)
```

The ith entry of remain_dims specifies whether the ith dimension is kept in the (TRUE) or is dropped (FALSE). Argument comm must be the communicator the original grid associated with. The new communicator, containing the subgrid that includes the calling process, returns in the argument **newcomm**.

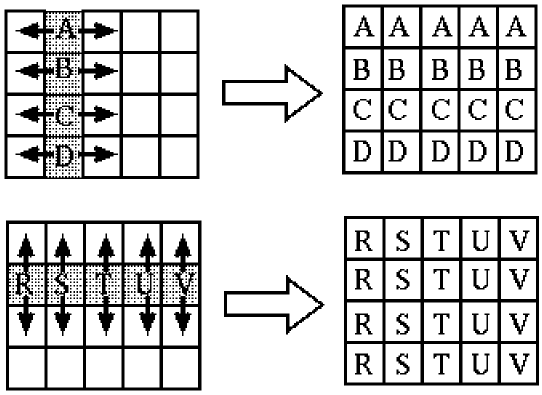

Figure 5-14: Performing broadcast operation along the first and the second dimension of a 2-D Cartesian grid.

For example, calling **MPI_Cart_sub** with `remain_dims=`(false,true) generates the necessary row subcommunicators for the first case in Figure 5-14. If `remain_dims=`(true,false) we get the subcommunicators for the second case in the figure.

However, partitioning mechanism can be applied in higher dimensions as well. If **comm** defines a 2x3x4 grid, and `remain_dims` = (TRUE, FALSE, TRUE), then **MPI_Cart_sub** will create three communicators each with eight processes in a 2x4 grid.

5.9.4 Example : circulating in a ring using Cartesian topology

To illustrate the usage of the Cartesian virtual topology, we are modified the source code of the example program described in Sect. 5.6.5. The modified version performs exactly the same task as the the original one: shift operations are taken place in a ring until each process receives its "starting value" back. (see Sect. 5.6.5 for more details, including the possible output of the program). The modified code is as follows:

[14] If the distance is 2, process ($x; y$) receives from ($x; y$ - 2) and sends to ($x; y$ + 2), etc.

```
#include "mpi.h"
#define NUM_DIMS 1

int main(argc, argv)
int argc;
char **argv;
{
    int myid, numprocs;
    MPI_Status status;
    MPI_Comm comm_cart;
    int dims[NUM_DIMS], periods[NUM_DIMS];
    int reorder; int datain, dataout;
    int source, dest;                    /* ranks of neighbours */
    int tag = 57;                        /* arbitrary value */

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

    /* set up parameters for MPI_Cart_create */
    dims[0] = numprocs;
    periods[0] = 1;                    /* true */
    reorder = 1;                       /* true */
    /* Make a new communicator with Cartesian topology */
    MPI_Cart_create( MPI_COMM_WORLD, NUM_DIMS,dims,
                     periods, reorder,&comm_cart);
    /*  Determine the ranks of the sending and
        receiving process for */
    /*  shift of 1 in the 0th dimension */
    MPI_Cart_shift(comm_cart, 0, 1, &source, &dest);

    datain = -1;
    dataout = myid;
    while (datain != myid) {
      /* now do the actual data shift */
      MPI_Sendrecv( &dataout, 1, MPI_INT, dest, tag,
                    &datain, 1, MPI_INT, source, tag,
                    comm_cart, &status);
      printf("Process %d received %d"n", myid, datain);
      dataout = datain;
    }

    /* We're at the end */
    MPI_Comm_free(&comm_cart);
    MPI_Finalize();
}
```

Each process performs the following main steps :

- After initializing the MPI, finds out the total number of processes and the own rank (**MPI_Init**, **MPI_Comm_size**, **MPI_Comm_rank**).
- Creates a new communicator (**comm_cart**) with a periodical one dimensional Cartesian grid (i.e. a simple ring topology), by calling **MPI_Cart_create**.
- Determines the ranks of the sender (**source**) and receiver (**dest**) processes for the shift of 1 in the Cartesian grid. Important to note that the communicator argument of **MPI_Cart_shift** has the actual value **comm_cart**. (I.e. it would be errornous to pass **MPI_COMM_WORLD** as actual parameter to this function because there is no topology information associated with the default communicator.)

- **MPI_Sendrecv** (see Sect. 5.6.3) performs the shift operation inside a `while loop` : it receives an integer value from **source** and store it into the variable **datain** and sends the value of **dataout** to the process with rank **destination**. Note that **comm_cart** must be passed as communicator argument to the sendrecv operation. After each shift, the process prints out a message to standard output.
- When the process gets back its own rank value (i.e. each rank value has gone full circle), the `while loop` ends and **MPI_Finalize** is called to terminate the MPI environment.

5.10 Topics not included

MPI is a large library. The Standard [MPI Forum94] is over 200 pages long and it defines more than 120 functions. Due to this fact, it is impossible to introduce all MPI facilities in such a short course. Topics which are not covered here include :

- persistent communication
- error handling
- profiling interface
- tools for writing libraries
- inter-communicators

Interested readers should refer to [MPI Forum94], [GropLusk94].

5.11 The future of MPI

The MPI-2 forum with old and new participants has begun a follow-on series of meetings. The following major topics are being considered:

- dynamic process management
- client/server
- real-time extensions
- "one-sided" communications (put/get, active messages)
- language bindings for C++ and Fortran-90