# Chapter 2
# Introduction to MPI: The Message Passing Interface

## 2.1 MPI for Parallel Programming: Communicating with Messages

Programming parallel algorithms is far more delicate than programming sequential algorithms. And so is debugging parallel programs too! Indeed, there exists several *abstract models* of "parallel machines" (parallel computations, distributed computations) with different kinds of *parallel programming paradigms*: For example, let us mention:

- *Vector super-computers* that rely on the programming model called *Single Instruction Multiple Data* (*SIMD*) with their optimized code based on pipelined operations,
- *Multi-core machines* with shared memory and their programming model using multi-threading, with all threads potentially accessing the shared memory. Programs can be easily crashing and it is difficult to debug sometimes due to potential conflicts when accessing concurrently a shared resource,
- *Clusters of computer machines* interconnected by a high-speed network that have a distributed memory.

It is precisely this last category of "parallel machines", the clusters of machines, that we are focusing on in this textbook: namely, parallel programming paradigm with distributed memory. Each computer can execute programs using its own local memory. Executed programs can be the same on all computers or can be different. Cooperation takes place by sending and receiving messages among these interconnected computers to accomplish their overall task.

Speaking on the size of these clusters, we can distinguish between:

- small-size to mid-size clusters of computers (say, dozens to hundreds, sometimes thousands, of computer nodes) that communicate with each other by sending and receiving messages, and
- large-size clusters (thousands to hundreds of thousands, sometimes millions computers) that execute rather simpler codes targeting *Big Data* processing.



F. Nielsen, *Introduction to HPC with MPI for Data Science*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-21903-5_2

Usually, these large-size clusters are programmed using the MapReduce/Hadoop programming model.

The Message Passing Interface (or MPI for short) standard is a programming interface: namely, an *Application Programming Interface* (*API*) that defines properly the syntax and full semantic of a software library that provides standardized basic routines to build complex programs thereof. Thus the MPI interface allows one to code parallel programs exchanging data by sending and receiving messages encapsulating those data. Using an API has the advantage of leaving the programmer free of the many details concerning the implementation of the fine details of implementing from scratch network procedures, and allows the ecosystem (academy, industry, programmers) to benefit of interoperability and portability of source codes. It is important to emphasize the fact that the MPI API *does not depend* on the underlying programming language it uses. Thus we can use MPI commands with the most common (sequential) programming languages like C, C++, Java, Fortran, Python and so on. That is, several *language bindings* of the MPI API are available.

MPI historically got initiated from a workshop organized in 1991 on distributed memory environments. Nowadays, we use the third version of the standard, *MPI-3*, which standardization has been completed and published openly in 2008. We shall choose *OpenMPI* (http://www.open-mpi.org/) to illustrate the programming examples in this book.

Let us emphasize that the MPI interface is the dominant programming interface for parallel algorithms with distributed memory in the HPC community. The strong argument in favor of MPI is the standardization of *many* (i) global routines of communication (like broadcasting, the routine that consists in sending a message to all other machines) and (ii) many primitives to perform global calculations (like computing a cumulative sum of data distributed among all machines using an aggregation mechanism). In practice, the complexity of these global communications and calculation operations depend on the underlying topology of the interconnection network of the machines of the cluster.

## 2.2 Parallel Programming Models, Threads and Processes

Modern operating systems are *multi-tasks*: from the user viewpoint, several non-blocking applications seem to be executed (run) "simultaneously". This is merely an illusion since on a single Central Processing Unit (CPU) there can be only one program instruction at a time being executed. In other words, on the CPU, there is a current process being executed while the others are blocked (suspended or waiting to be waked up) and wait their turn to be executed on the CPU. It is the rôle of the *task scheduler* to allocate dynamically processes to CPU.

Modern CPUs have several *cores* that are independent *Processing Units* (PUs) that can execute truly in parallel on each core a thread. Multi-core architectures yield the multi-threading programming paradigm that allows for *concurrency*. For example,

your favorite Internet WEB browser allows you to visualize simultaneously several pages in their own tabs: each HTML page is rendered using an independent thread that retrieves from the network the page contents in HTML[1]/XML and displays it. The resources allocated to a process are *shared* between the different threads, and at least one thread should have a `main` calling function.

We can characterize the threads as follows:

- Threads of a same process share the same memory area, and can therefore access both the data area but also the code area in memory. It is therefore easy to access data between the threads of a same process, but it can also raise some difficulties in case of simultaneous access to the memory: In the worst case, it yields a system crash! A theoretical abstraction of this model is the so-called *Parallel Random-Access Machine* (or *PRAM*). On the PRAM model, we can classify the various conflicts that can happen when reading or writing simultaneously on the local memory. We have the *Exclusive Read Exclusive Write* sub-model (EREW), the *Concurrent Read Exclusive Write* (CREW) and the *Concurrent Read Concurrent Write* models.
- This multi-threading programming model is very well suited to multi-core processors, and allows applications to be ran faster (for example, for encoding a MPEG4 video or a MP3 music file) or using non-blocking applications (like a web multi-tab browser with a mail application).
- Processes are different from threads because they have their own *non-overlapping* memory area. Therefore, communications between processes have to be done careful, in particular using the MPI standard.

We can also distinguish the parallel programming paradigm *Single Program Multiple Data* (SPMD) from the paradigm called Multiple Program Multiple Data (MPMD). Finally, let us notice that we can run several processes either on a same processor (in parallel when the processor is multi-core) or on a set of processors interconnected by a network. We can also program processes to use several multi-core processors (in that case, using both the MPI and OpenMP standards).

## 2.3 Global Communications Between Processes

By executing a MPI program on a cluster of machines, we launch a set of processes, and we have for each process traditional local computations (like ordinary sequential programs) but also:

- some data transfers: for example, some data broadcasted to all other processes using a message,
- some synchronization barriers where all processes are required to wait for each other before proceeding,

[1] *Hypertext Markup Language*.

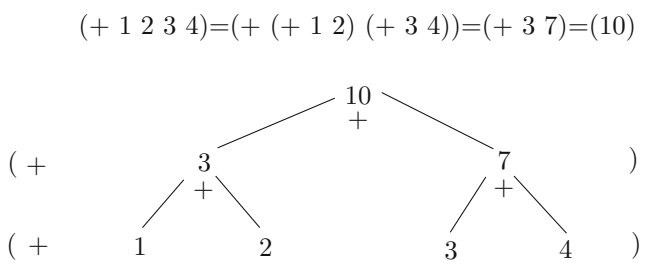


**Fig. 2.1** Example of a global reduce computation: calculating the global cumulative sum of the local values of a process variable. We illustrate here the reduction tree performed when invoking the reduce primitive

- global computations: for example, a reduce operation that calculates, say, the sum or the minimum of a distributed variable $x$ belonging to all the processes (with a local value stored on the local memory of each process). Figure 2.1 illustrates a reduce cumulative sum operation. The global computation depends on the underlying topology of the interconnected cluster machines.

Global communication primitives are carried out on all processes belonging to the same *group of communication*. By default, once MPI got initialized, all processes belong to the same group of communication called `MPI_COMM_WORLD`.

### 2.3.1 Four Basic MPI Primitives: Broadcast, Gather, Reduce, and Total Exchange

The MPI broadcasting primitive, `MPI_Bcast`, sends a message from a *root process* (the calling process of the communication group) to all other processes (belonging to the communication group). Conversely, the reduce operation aggregates all corresponding values of a variable into a single value that is returned to the calling process. When a different personalized message is send to each other process, we get a scatter operation called in MPI by `MPI_Scatter`.

Aggregating primitives can either be for communication of for computing globally: gather is the converse of the scatter operation where a calling process receives from all other processes a personalized message. In MPI, `MPI_Reduce` allows one to perform a *global calculation* by aggregating (reducing) the values of a variable using a commutative binary operator.[2] Typical such examples are the cumulative sum (`MPI_SUM`) or the cumulative product (`MPI_PROD`), etc. A list of such binary operators used in the reduce primitives is given in Table 2.1. Those four basic MPI primitives are illustrated in Fig. 2.2. Last but not least, we can also call a global

[2] An example of binary operator that is not commutative is the division since $p/q \neq q/p$.

**Table 2.1** Global calculation: predefined (commutative) binary operators for MPI reduce operations

| MPI name | Meaning |
|---|---|
| `MPI_MAX` | Maximum |
| `MPI_MIN` | Minimum |
| `MPI_SUM` | Sum |
| `MPI_PROD` | Product |
| `MPI_LAND` | Logical and |
| `MPI_BAND` | Bit-wise and |
| `MPI_LOR` | Logical or |
| `MPI_BOR` | Bit-wise or |
| `MPI_LXOR` | Logical xor |
| `MPI_BXOR` | Bit-wise xor |
| `MPI_MAXLOC` | Max value and location |
| `MPI_MINLOC` | Min value and location |

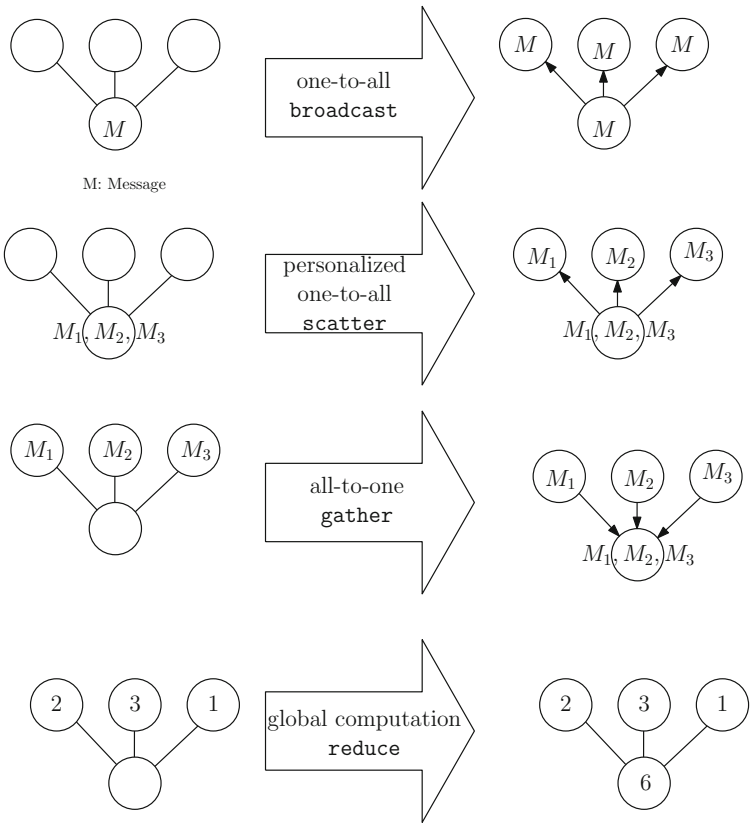


**Fig. 2.2** Four basic collective communication primitives: broadcast (one to all), scatter (personalized broadcast or personalized one-to-all communication), gather (the inverse of a scatter primitive, or personalized all-to-one communication) and reduce (a global collective computation)

all-to-all communication primitive (`MPI_Alltoall`, also called *total exchange*) where each process sends a personalized message to all other processes.

### 2.3.2 Blocking Versus Non-blocking and Synchronous Versus Asynchronous Communications

MPI has several send modes depending on whether data are buffered or not and whether synchronization is required or not. First, let us start by denoting by `send` and `receive` the two basic communication primitives. We describe the syntax and semantic of these two primitives as follows:

- `send(&data, n, Pdest)`: Send an array of *n* data starting at memory address `&data` to process `Pdest`
- `receive(&data, n, Psrc)`: Receive *n* data from process `Psrc` and store them in an array which starts a local memory address &data

Now, let us examine what happens in this example below:

| Process P0 | Process P1 |
|---|---|
| `...` | `...` |
| `a=442;` | `receive(&a, 1, P0);` |
| `send(&a, 1, P1);` | `cout << a << endl;` |
| `a=0;` | |

Blocking communications (not buffered) yield a *waiting time* situation: that is, a *idling time*. Indeed, the sending process and the receiving process need to wait mutually for each other: it is the communication mode commonly termed hand-shaking. This mode allows one to perform synchronous communications. Figure 2.3 illustrates these synchronous communications by hand-shaking and indicates the idling periods.

The C program below gives an elementary example of blocking communication in MPI (using the *C binding* of the OpenMPI vendor implementation of MPI):

WWW source code: `MPIBlockingCommunication.cpp`

```
// filename: MPIBlockingCommunication.cpp
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>

int main(argc,argv)
int argc;
char *argv[];
{
     int myid, numprocs;
```

```
    int tag,source,destination,count;
    int buffer;
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    tag=2312; /* any integer to tag messages */
    source=0;
    destination=1;
    count=1;
    if(myid == source){
      buffer=2015;
      MPI_Send(&buffer,count,MPI_INT,destination,tag
          ,MPI_COMM_WORLD);
      printf("processor %d received %d \n", myid,
          buffer)
    }
    if(myid == destination){
        MPI_Recv(&buffer,count,MPI_INT,source,tag,
            MPI_COMM_WORLD,&status);
        printf("processor %d received %d \n",myid,
            buffer);
    }
    MPI_Finalize();
}
```

Clearly, for blocking communications, one seeks to minimize the overall idling time. We shall see later on how to perform this optimization using a load-balancing technique to balance fairly the local computations among the processes.

We report the syntax and describe the arguments of the `send`[3] primitive in MPI:

- Syntax using the C binding:

```
#include <mpi.h>
int MPI_Send(void *buf, int count, MPI_Datatype
    datatype, int dest, int tag, MPI_Comm comm)
```

- Syntax in C++ (Deprecated. That is, it is not regularly updated since MPI-2 and we do not recommend using it):

```
#include <mpi.h>
void Comm::Send(const void* buf, int count, const
    Datatype& datatype, int dest, int tag) const
```

The `tag` argument in `send` assigns an integer to a message (its label) so that processes can specify which type of message to wait for. Tags are useful in practice to filter communication operations, and ensures for example that messages sent/received are pairwise matching using blocking communications.

The C data types in MPI are summarized in Table 2.2.

[3] See the manual online: https://www.open-mpi.org/doc/v1.4/man3/MPI_Send.3.php.

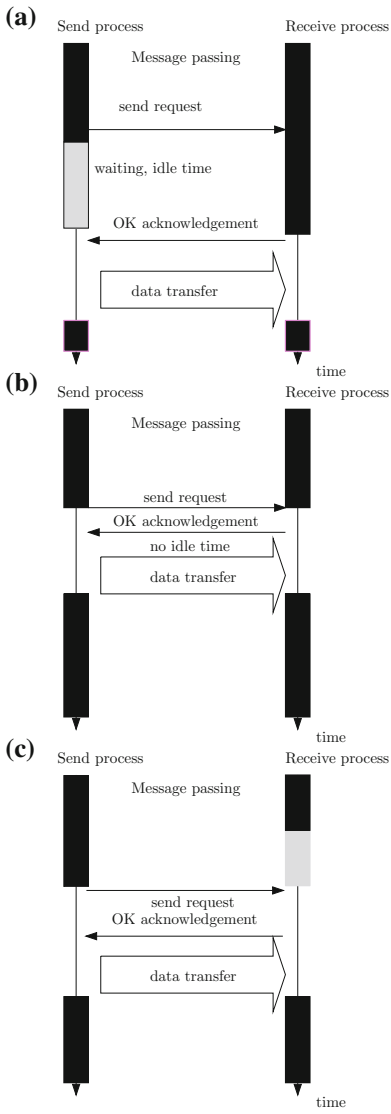


**Fig. 2.3** Blocking communications by hand-shaking: **a** sending process waits for the "OK" of the receiver and thus provoke a waiting situation, **b** one seeks to minimize this idling time, and **c** case where it is the receiving process that needs to wait

### *2.3.3 Deadlocks from Blocking Communications*

Using blocking communications allows one to properly match "send queries" with "receive queries", but can unfortunately also yields deadlocks.[4]

[4]In that case, either a time-out signal can be emitted externally to kill all the processes, or we need to manually kill the processes using their process identity number using Shell command line instructions.

**Table 2.2** Basic data types in MPI when using the C language binding

| MPI type | Corresponding type in the C language |
|---|---|
| `MPI_CHAR` | `signed char` |
| `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` |
| `MPI_BYTE` | |
| `MPI_PACKED` | |

Let us consider this toy example to understand whether there is a deadlock situation occurring or not:

| Process P0 | Process P1 |
|---|---|
| `send(&a, 1, P1);` | `send(&a, 1, P0);` |
| `receive(&b, 1, P1);` | `receive(&b, 1, P0);` |

Process $P0$ send a message and then waits the green light "OK for sending" of receiver process $P1$, but the send query of $P1$ also waits for the green light "OK for sending" of process $P0$. That is typically a *deadlock* situation. In this toy example, we have highlighted the deadlock problem when using blocking communication primitives. However in practice, it is not so easy to track them as process programs can take various execution paths.

In practice, in MPI, each send/receive operation concerns a group of communication and has a tag attribute (an integer). From an algorithmic viewpoint, blocking communications are a highly desirable feature to ensure consistency (or the semantic) of programs (for example, to avoid that messages arrive in wrong orders) but they can yield difficulties to detect deadlocks.

In order to remove (or at least minimize!) these deadlock situations, we can preallocate to each process a dedicated memory space for buffering data: the *data buffer* (bearing the acronym DB). We then send data in two steps:

- First, the send process sends a message on the data buffer, and
- Second, the receive process copies the data buffer on its local memory area indicated by the address `&data`.

This buffered communication can either be implemented by hardware mechanisms or by appropriate software. However, there still remains potential deadlocks when the data buffers become full (raising a "buffer overflow" exception). Even if

we correctly manage the `send` primitives, there can still be remaining deadlocks, even with buffered communications, because of the blocking receive primitive. This scenario is illustrated as follows:

| Process P0 | Process P1 |
|---|---|
| `receive(&a, 1, P1);` | `receive(&a, 1, P0);` |
| `send(&b, 1, P1);` | `send(&b, 1, P0);` |

Each process waits for a message before being able to send its message! Again, this is a deadlock state! In summary, blocking communications are very useful when we consider global communication like broadcasting in order to ensure the correct arrival order of messages, but one has to take care of potential deadlocks when implementing these communication algorithms.

A solution to avoid deadlocks is to consider both the `send` and `receive` primitives being non-blocking. These *non-blocking communication routines* (not buffered) are denoted by `Isend` and `Ireceive` in MPI: There are *asynchronous communications*. In that case, the send process posts a message "Send authorization request" (a pending message) and continues the execution of its program. When the receiver process posts a "OK for sending" approval, data transfers are initiated. All these mechanics are internally managed using signals of the operating system. When the data transfer is completed, a check status let indicate whether processes can proceed to read/write data safely. The C program below illustrates such a non-blocking communication using the C binding of OpenMPI. Let us notice that the primitive `MPI_Wait(&request,&status);` waits until the transfer is completed (or interrupted) and indicates whether that transfer has been successful or not, using a state variable called `status`.

WWW source code: `MPINonBlockingCommunication.cpp`

```
// filename: MPINonBlockingCommunication.cpp
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>

int main(argc,argv)
int argc;
char *argv[];
{
     int myid, numprocs;
     int tag,source,destination,count;
     int buffer;
     MPI_Status status;
     MPI_Request request;

     MPI_Init(&argc,&argv);
     MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
     MPI_Comm_rank(MPI_COMM_WORLD,&myid);
     tag=2312;
```

```
      source=0;
      destination=1;
      count=1;
      request=MPI_REQUEST_NULL;
      if(myid == source){
        buffer=2015;
        MPI_Isend(&buffer,count,MPI_INT,destination,
            tag,MPI_COMM_WORLD,&request);
      }
      if(myid == destination){
          MPI_Irecv(&buffer,count,MPI_INT,source,tag,
              MPI_COMM_WORLD,&request);
      }
      MPI_Wait(&request,&status);
      if(myid == source){
        printf("processor %d  sent %d\n",myid,buffer);
      }
      if(myid == destination){
        printf("processor %d  received %d\n",myid,
            buffer);
      }
      MPI_Finalize();
}
```

We summarize the calling syntax in the C binding of the non-blocking primitives `Isend` and `Irecv`:

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

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

The structure `MPI_Request` is often used in programs: it returns `*flag=1` when the operation `*req` has completed, and `0` otherwise.

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

The primitive MPI_Wait waits until the operation indicated by `*req` has been completed

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

We summarize the various communication protocols (blocking/non-blocking send with blocking/non-blocking receive) in Table 2.3.

**Table 2.3** Comparisons of the various send/receive operation protocols

| | Blocking operation | Non-blocking operation |
|---|---|---|
| Bufferized | `send` completes after data have been copied to the data buffer | `send` completes after having initialized DMA (*Direct Memory Access*) transfer to the data buffer. The operation is not necessarily completed after it returns |
| Not-bufferized | Blocking `send` until it meets a corresponding `receive` | To define |
| Meaning | Semantic of `send` and `receive` by matching operations | Semantic must be explicitly specified by the programmer that needs to check the operation status |

The program listings so far highlighted eight common procedures of MPI (among a rich set of MPI instructions):

| | |
|---|---|
| `MPI_Init` | Initialize the MPI library |
| `MPI_Finalize` | Terminate MPI |
| `MPI_Comm_size` | Return the number of processes |
| `MPI_Comm_rank` | Rank of the calling process |
| `MPI_Send` | send a message (blocking) |
| `MPI_Recv` | receive message (blocking) |
| `MPI_Isend` | send a message (non-blocking) |
| `MPI_Irecv` | receive message (non-blocking) |

All these procedures return `MPI_SUCCESS` when they are completed with success, or otherwise an error code depending on the problems. Data types and constants are prefixed with `MPI_` (we invite the reader to explore the header file `mpi.h` for more information).

### 2.3.4 Concurrency: Local Computations Can Overlap with Communications

It is usual to assume that the processors (or Processing Elements, PEs) can perform several tasks at the same time: for example, a typical scenario is to use non-blocking communications (`MPI_IRecv` and `MPI_ISend`) at the same time they perform some local computations. Thus we require that those three operations are not interferring with each other. In one stage, we can therefore not send the result of a calculation and we can not send what has been concurrently received (meaning

forwarding). In parallel algorithmic, we denote by the double vertical bar || these concurrent operations:

```
IRecv||ISend||Local_Computation
```

### 2.3.5 Unidirectional Versus Bidirectional Communications

We distinguish between *one-way communication* and *two-way communication* as follows: in one-way communication, we authorize communications over communication channels in one direction only: that is, either we send a message or we receive a message (`MPI_Send`/`MPI_Recv`) but not both at the same time. In a two-way communication setting, we can communicate using both directions: in MPI, this can be done by calling the procedure MPI_Sendrecv.[5]

### 2.3.6 Global Computations in MPI: Reduce and Parallel Prefix (Scan)

In MPI, one can perform global computations like the cumulative sum $V = \sum_{i=0}^{P-1} v_i$ where $v_i$ is a local variable stored in the memory of process $P_i$ (or the cumulative product $V = \prod_{i=0}^{P-1} v_i$). The result of this global computation $V$ is then available in the local memory of the process that has called this reduce primitive: the calling process, also called the root process. We describe below the usage of the `reduce`[6] primitive using the C binding of OpenMPI:

```
#include <mpi.h>

int MPI_Reduce ( // Reduce routine
 void* sendBuffer, // Address of local val
 void* recvBuffer, // Place to receive into
 int count, // No. of elements
 MPI_Datatype datatype, // Type of each element
 MPI_OP op, // MPI operator
 int root, // Process to get result
 MPI_Comm comm // MPI communicator
 );
```

Reduction operations are predefined and can be selected using one of the keywords among this list (see also Table 2.1).

[5] https://www.open-mpi.org/doc/v1.8/man3/MPI_Sendrecv.3.php.

[6] See manual online at https://www.open-mpi.org/doc/v1.5/man3/MPI_Reduce.3.php.

| | |
|---|---|
| `MPI_MAX` | maximum |
| `MPI_MIN` | minimum |
| `MPI_SUM` | sum $\sum$ |
| `MPI_PROD` | product $\prod$ |
| `MPI_LAND` | logic AND |
| `MPI_BAND` | bitwise AND |
| `MPI_LOR` | logic OR |
| `MPI_BOR` | bitwise OR |
| `MPI_LXOR` | logic XOR |
| `MPI_BXOR` | bitwise COR |
| `MPI_MAXLOC` | maximal value and corresponding index of the maximal element |
| `MPI_MINLOC` | minimal value and corresponding index of the minimal element |

In MPI, one can also build its own data type and define the associative and commutative binary operator for reduction.

A second kind of global computation are *parallel prefix* also called *scan*. A `scan` operation calculates all the partial reductions on the data stored locally on the processes.

Syntax in MPI is the following:

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

Calling this procedure allows one to perform a *prefix reduction* on the data located in `sendbuf` on each process with the result available in the memory address `recvbuf`. Figure 2.4 illustrates graphically the difference between these two global computation primitives: `reduce` and `scan`.

```
MPI_Scan( vals, cumsum, 4, MPI_INT, MPI_SUM,
    MPI_COMM_WORLD )
```

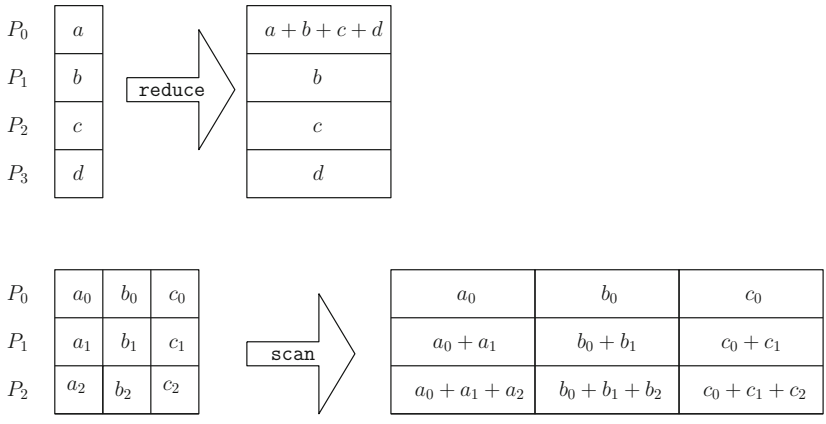


**Fig. 2.4** Visualizing a reduce operation and a parallel prefix (or scan) operation

We described by syntax of `reduce` and `scan` using the C binding because the C++ binding is not any longer updated since MPI-2. In practice, we often program using the modern oriented-object C++ language and call the MPI primitives using the C interface. Recall that the C language [1] is a precursor of C++ [2] that is *not* an oriented-object language, and manipulates instead data structures defined by the keyword `struct`.

These global computations are often implemented internally using spanning trees of the underlying topology of the interconnection network.

### 2.3.7 Defining Communication Groups with Communicators

In MPI, communicators allow one to group processes into various groups of communications. Each process is included in a communication and is indexed by its *rank* inside this communication group. By default, `MPI_COMM_WORLD` includes all the $P$ processes with the rank being an integer ranging from 0 to $P-1$. To get the number of processes inside its communication group or its rank inside the communication, we use the following primitives in MPI: `int MPI_Comm_size` (MPI_Comm comm, int *size) and `int MPI_Comm_rank` (MPI_Comm comm, int *size).

For example, we create a new communicator by removing the first process as follows:

WWW source code: `MPICommunicatorRemoveFirstProcess.cpp`

```
// filename: MPICommunicatorRemoveFirstProcess.cpp
#include <mpi.h>

int main(int argc, char *argv[])
    {
    MPI_Comm comm_world, comm_worker;
    MPI_Group group_world, group_worker;
    comm_world = MPI_COMM_WORLD;

    MPI_Comm_group(comm_world, &group_world);
    MPI_Group_excl(group_world, 1, 0, &group_worker)
        ;

    /* process 0 is removed from the communication
        group */

    MPI_Comm_create(comm_world, group_worker, &
        comm_worker);
}
```

In this second listing, we illustrate how to use communicators:

WWW source code: `MPICommunicatorSplitProcess.cpp`

```
// filename: MPICommunicatorSplitProcess.cpp
#include <mpi.h>
#include <stdio.h>
#define NPROCS 8

int main(int argc, char *argv[])
    {
    int *ranks1[4]={0,1,2,3}, ranks2[4]={4,5,6,7};

    MPI_Group orig_group, new_group;
    MPI_Comm new_comm

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    sendbuf = rank;

    // Retrieve the intial group
    MPI_Comm_group(MPI_COMM_WORLD, &orig_group);

    if (rank < NPROCS/2)
        MPI_Group_incl(orig_group, NPROCS/2, ranks1,
            &new_group);
 else
    MPI_Group_incl(orig_group, NPROCS/2, ranks2, &
       new_group);

// create new communicator
MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm
    );

// global computation primitive
MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT,
    MPI_SUM, new_comm);
MPI_Group_rank (new_group, &new_rank);
printf("rank= %d newrank= %d recvbuf= %d\n", rank,
    newrank, recvbuf);

MPI_Finalize();
}
```

`MPI_Comm_create` is a collective operation. All processes of the former communication group need to call it, even those who do not belong to the new communication group.

## 2.4 Synchronization Barriers: Meeting Points of Processes

In the coarse-grained parallelism mode, processes execute large chunks of computations independently from each other. Then they wait for each other at a *synchronization barrier* (see Fig. 2.5, `MPI_Barrier` in MPI), perform some send/receive messages, and proceed their program execution.

### *2.4.1 A Synchronization Example in MPI: Measuring the Execution Time*

For example, let us illustrate a way to measure the parallel time of a MPI program with a synchronization barrier. We shall use the procedure `MPI_Wtime` to measure time in MPI. Consider this master/slave code:

WWW source code: `MPISynchronizeTime.cpp`

```
// filename: MPISynchronizeTime.cpp
double start, end;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

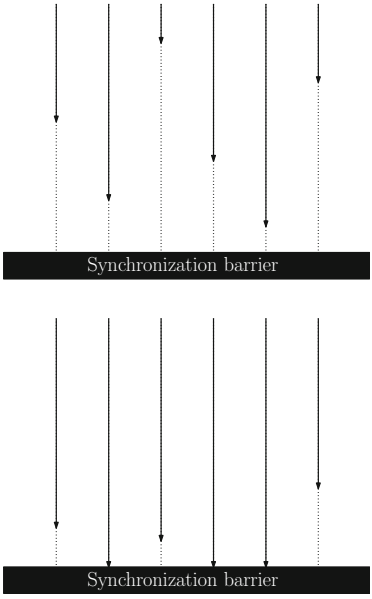


**Fig. 2.5** Conceptual illustration of a synchronization barrier: processes wait for each other at a synchronization barrier before carrying on the program execution

```
MPI_Barrier(MPI_COMM_WORLD); /* IMPORTANT */
start = MPI_Wtime();   .

/* some local computations here */
LocalComputation();

MPI_Barrier(MPI_COMM_WORLD); /* IMPORTANT */
end = MPI_Wtime(); /* measure the worst-case time of
     a process */

MPI_Finalize();

if (rank == 0)
         { /* use time on master node */
      cout<< end-start <<endl; // here we use C++ syntax
         }
```

We can also use a `MPI_Reduce()` procedure to compute the minimum, maximum, and overall sum of all the process times. But this eventually requires to add an extra step for perform the global computation with a reduce operation.

### 2.4.2 The Bulk Synchronous Parallel (BSP) Model

One of the high-level parallel programming model is called the *Bulk Synchronous Parallel* (or *BSP* for short). This abstract model has been conceived by Leslie G. Valiant (Turing award, 2010) and facilitates the design of parallel algorithms using three fundamental steps that form a "super-step":

1. concurrent computation step: processes locally and asynchronously compute, and those local computation can overlap with communications,
2. communication step: processes exchange data between themselves,
3. synchronization barrier step: when a process reaches a synchronization barrier, it waits for all the other processes to reach this barrier before proceeding another super-step.

A parallel algorithm on the BSP model is a sequence of super-steps. A software library, `BSPonMPI`,[7] allows one to use this programming model easily with MPI.

[7] http://bsponmpi.sourceforge.net/.

## 2.5 Getting Started with the MPI: Using OpenMPI

We describe several ways to use the OpenMPI implementation of the MPI standard using either the C, C++, or Boost bindings. There is also a convenient Python binding[8] that is touched upon.

### 2.5.1 *The "Hello World" Program with MPI C++*

The traditional "Hello program" reflects the minimal structure of a program displaying a simple message.

WWW source code: `MPIHelloWorld.cpp`

```
// filename: MPIHelloWorld.cpp
# include <iostream>
using namespace std;
# include "mpi.h"
int main ( int argc, char *argv[] )
{
  int id, p, name_len;
  char processor_name[MPI_MAX_PROCESSOR_NAME];
// Initialize MPI.
  MPI::Init ( argc, argv );
// Get the number of processes.
  p = MPI::COMM_WORLD.Get_size ( );
// Get the individual process ID.
  id = MPI::COMM_WORLD.Get_rank ( );
  MPI_Get_processor_name(processor_name, &name_len);
  // Print off a hello world message
 cout << "  Processor " << processor_name<<"  ID="<<
    id << " Welcome to MPI!'\n";
// Terminate MPI.
  MPI::Finalize ( );
return 0;
}
```

To compile this C++ source code, we type in the terminal:

```
mpic++ welcomeMPI.cpp -o welcomeMPI
```

When the `-o` option is not set, the compiler will write the byte code in a default file called: `a.out`. Once compile, we execute this program, here on machine named `machinempi`:

```
>$ mpirun -np 4 welcomeMPI
  Processor machinempi  ID=3 Welcome MPI!'
```

[8] http://mpi4py.scipy.org/docs/usrman/.

```
  Processor machinempi  ID=0 Welcome MPI!'
  Processor machinempi  ID=1 Welcome MPI!'
  Processor machinempi  ID=2 Welcome MPI!'
```

Let us note that on the console, the messages are displayed in the order of the execution time of the `cout` orders. Thus if we launch again this program, we order of messages on the console may be different. Therefore let us emphasize that when we invoke the `mpirun` command, we create $P$ processes that all execute the same compiled code. Each process can take different branches of the program by identifying themselves using their rank.

We can use two machines to run the program as follows:

```
>$ mpirun -np 5 -host machineMPI1,machineMPI2 welcomeMPI
  Processor machineMPI2  ID=1 Welcome MPI!'
  Processor machineMPI2  ID=3 Welcome MPI!'
  Processor machineMPI1  ID=0 Welcome MPI!'
  Processor machineMPI1  ID=2 Welcome MPI!'
  Processor machineMPI1  ID=4 Welcome MPI!'
```

The `mpirun` execution command is a symbolic link to the `orterun` command in OpenMPI. We can list the various libraries of MPI as follows:

```
>mpic++ --showme:libs
mpi_cxx mpi open-rte open-pal dl nsl util m dl
```

And we can add a new library as follows:

```
export LIBS=${LIBS}:/usr/local/boost-1.39.0/include/boost-1_39
```

And then compile this command line in the shell:

```
mpic++ -c t.cpp -I$LIBS
```

As usual, it is better to set the shell configuration file properly by editing the `.bashrc`. After having done your editing, you need to re-read the configuration file by typing this built-in-shell command:

```
source ~/.bashrc
```

You are now ready to use simultaneously a large number of machines. But please keep in mind that you have to be kind when using a large number of resources: everybody shall uses the shared resource fairly!

In the Appendix B, we describe the SLURM task scheduler to launch MPI jobs on a cluster of machines.

### 2.5.2 Programming MPI with the C Binding

The toy program below describes a way to define a master-slave program:

WWW source code: `MPICBindingExample.c`

```
/* filename: MPICBindingExample.c */
int main (int argc, char **argv)
{
        int myrank, size;
        MPI_Init (&argc, &argv);
        MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
        MPI_Comm_size( MPI_COMM_WORLD, &size );

        if (!myrank)
          master ();
        else
                  slave ();
        MPI_Finalize ();
        return (1);
}

void master()
        {printf("I am the master program\n");}

void slave()
        {printf("I am the slave program\n");}
```

Note that the MPI interface that we have used here is fairly different from our first program "Hello World". Indeed, we use the C binding of the MPI here. The C binding is the most commonly used binding of MPI and is frequently updated. It offers all functions of the MPI standard. The C++ binding is not anymore supported and offers less functions. Therefore we recommend to use the C binding, even in a C++ program (C calling style of MPI procedures inside a C++ object-oriented program). This explains why in our codes, we have C calling function style with `cout` print order to the console!

### 2.5.3 Using MPI with C++ Boost

*Boost*[9] is a C++ library that is very useful for dealing with matrices and graphs, etc. Interesting, this library also offers its own style to use MPI programs. Here is a small Boost-MPI program to showcase the library:

[9] http://www.boost.org/.

WWW source code: `MPIBoostBindingExample.cpp`

```
// filename: MPIBoostBindingExample.cpp
#include <boost/mpi/environment.hpp>
#include <boost/mpi/communicator.hpp>
#include <iostream>
namespace mpi = boost::mpi;

int main()
{
    mpi::environment env;
    mpi::communicator world;
    std::cout << "I am process " << world.rank() << "
         on " << world.size()
              << "." << std::endl;
    return 0;
}
```

If you are using Unix, you can compile this program as follows:

```
/usr/local/openmpi-1.8.3/bin/mpic++ -I/usr/local/boost-1.56.0/include/
-L/usr/local/boost-1.56.0/lib/ -lboost_mpi -lboost_serialization myprogram.cpp
-o myprogram
```

## 2.6 Using MPI with OpenMP

*OpenMP*[10] is yet another Application Programming Interface for parallel programming with shared memory. OpenMP is a cross-platform standard that offers bindings in the C/C++/Fortran imperative languages among others. OpenMP is typically used when one wants to use multi-core processors. Here is a "Hello World" program using both the MPI and OpenMP APIs:

WWW source code: `MPIOpenMPExample.cpp`

```
// filename: MPIOpenMPExample.cpp
#include <mpi.h>
#include <omp.h>
#include <stdio.h>
int main (int nargs, char** args)
{
int rank, nprocs, thread_id, nthreads;
int name_len;
char processor_name[MPI_MAX_PROCESSOR_NAME];

MPI_Init (&nargs, &args);
```

[10] http://openmp.org/wp/.

```
MPI_Comm_size (MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
MPI_Get_processor_name(processor_name, &name_len);

#pragma omp parallel private(thread_id, nthreads)
{
thread_id = omp_get_thread_num ();
nthreads = omp_get_num_threads ();
printf("Thread number %d (on %d) for the MPI process
     number %d (on %d) [%s]\n",
thread_id, nthreads, rank, nprocs,processor_name);
}
MPI_Finalize ();
return 0;
}
```

We use the option `-fopenmp` of the mpic++ compiler as follows:

```
mpic++ -fopenmp testmpiopenmp.cpp -o testmp.exe
```

Then we execute this program at the command line as follows:

```
mpirun -np 2 -host royce,simca testmp.exe
```

```
[royce ~]$ mpirun -np 2 -host royce,simca dmp.exe
Thread number 0 (on 8) for the MPI process number 1 (on 2) [simca.polytechnique.fr]
Thread number 1 (on 8) for the MPI process number 1 (on 2) [simca.polytechnique.fr]
Thread number 5 (on 8) for the MPI process number 1 (on 2) [simca.polytechnique.fr]
Thread number 4 (on 8) for the MPI process number 1 (on 2) [simca.polytechnique.fr]
Thread number 3 (on 8) for the MPI process number 1 (on 2) [simca.polytechnique.fr]
Thread number 7 (on 8) for the MPI process number 1 (on 2) [simca.polytechnique.fr]
Thread number 0 (on 8) for the MPI process number 0 (on 2) [royce.polytechnique.fr]
Thread number 1 (on 8) for the MPI process number 0 (on 2) [royce.polytechnique.fr]
Thread number 5 (on 8) for the MPI process number 0 (on 2) [royce.polytechnique.fr]
Thread number 4 (on 8) for the MPI process number 0 (on 2) [royce.polytechnique.fr]
Thread number 7 (on 8) for the MPI process number 0 (on 2) [royce.polytechnique.fr]
Thread number 2 (on 8) for the MPI process number 0 (on 2) [royce.polytechnique.fr]
Thread number 3 (on 8) for the MPI process number 0 (on 2) [royce.polytechnique.fr]
Thread number 6 (on 8) for the MPI process number 0 (on 2) [royce.polytechnique.fr]
Thread number 2 (on 8) for the MPI process number 1 (on 2) [simca.polytechnique.fr]
Thread number 6 (on 8) for the MPI process number 1 (on 2) [simca.polytechnique.fr]
```

It can be seen that the two host machines have 8 cores each. We observe that the arrival printing order on the console depends on many system factors. Running another time the program will likely yield a different arrival order. Instead of naming explicitly the host machines, we can also use a resource scheduler like SLURM[11] that will allocate automatically all necessary resources of a cluster to MPI programs (see Appendix B).

### 2.6.1 Programming MPI with the Python Binding

*Python*[12] has become widely popular the last decade as a fast prototyping language. The Python binding is available from the following URL: http://mpi4py.scipy.org/docs/usrman/

[11] https://computing.llnl.gov/linux/slurm/.

[12] https://www.python.org/.

WWW source code: `MPIHelloWorld.py`

```
#!/usr/bin/env python
"""
MPI Hello World example
"""

from mpi4py import MPI
import sys

size = MPI.COMM_WORLD.Get_size()
rank = MPI.COMM_WORLD.Get_rank()
name = MPI.Get_processor_name()

sys.stdout.write(
    "Hello, World! I am process %d of %d on %s.\n"
    % (rank, size, name))
```

```
# mpirun -np 5 python26 hw.py
```

## 2.7 Main Primitives in MPI

We recall the main collective communication primitives that are global operations performed on a communicator (group of machines):

- broadcast (one-to-all) and reduce (all-to-one, that can be interpreted as the reverse operation of a broadcast primitive),
- scatter or personalized broadcast that sends different messages to all processes,
- gather or all-to-one that assembles individual messages from all processes to the calling process (inverse operation of a scatter primitive)
- global computational like reduce or scan (as known as parallel prefix),
- total communication, all-to-all, also called total exchange (personalized messages for all processes),
- etc.

### *2.7.1 MPI Syntax for Broadcast, Scatter, Gather, Reduce and Allreduce*

- broadcast: `MPI_Bcast`[13]

```
int MPI_Bcast(void *buffer, int count,
```

[13] https://www.open-mpi.org/doc/v1.5/man3/MPI_Bcast.3.php.

```
MPI_Datatype datatype,
int root, MPI_Comm comm)
```

- scatter: `MPI_Scatter`[14]

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

- gather: `MPI_Gather`[15]

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

- reduce: `MPI_Reduce`[16]

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

- Allreduce: `MPI_Allreduce`[17]

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

Those operations are visually explained in Fig. 2.6.

### 2.7.2 *Other Miscellaneous MPI Primitives*

- The *prefix sum* primitive considers a binary associative operator $\oplus$ like $+$, $\times$, max, min, and computes for $0 \leq k \leq P-1$ the "sum" stored on $P_k$:

$$S_k = M_0 \oplus M_1 \oplus \cdots \oplus M_k$$

The $P$ messages $\{M_k\}_k$ are assumed to be stored on the local memory of process $P_k$.

- *all-to-all reduce* is defined according to a binary associative operation $\oplus$ like $+$, $\times$, max, min, and outputs:

$$M_r = \oplus_{i=0}^{P-1} M_{i,r}.$$

[14] https://www.open-mpi.org/doc/v1.5/man3/MPI_Scatter.3.php.
[15] https://www.open-mpi.org/doc/v1.5/man3/MPI_Gather.3.php.
[16] https://www.open-mpi.org/doc/v1.5/man3/MPI_Reduce.3.php.
[17] https://www.open-mpi.org/doc/v1.5/man3/MPI_Allreduce.3.php.

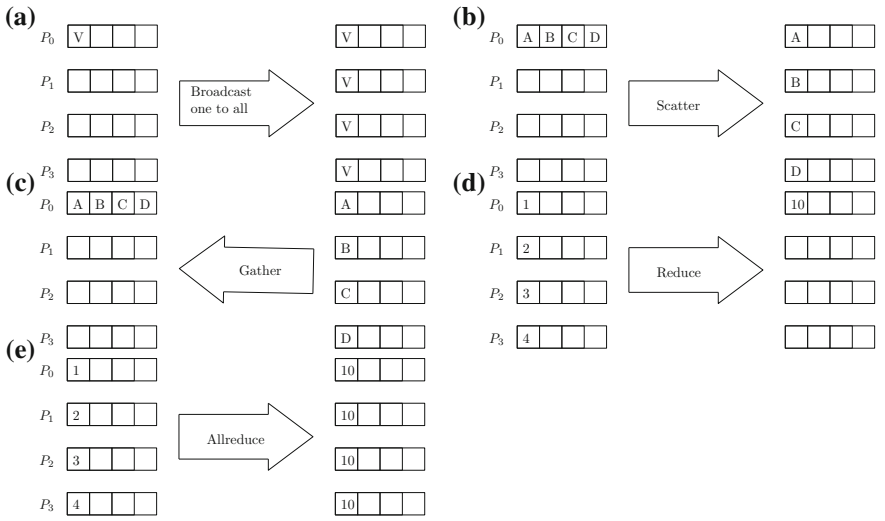


**Fig. 2.6** Standard communication primitives and global computations in MPI: **a** broadcast, **b** scatter, **c** gather, **d** reduce and **e** allreduce

We have $P^2$ messages $M_{r,k}$ for $0 \leq r, k \leq P-1$, and messages $M_{r,k}$ are stored locally on $P_r$.

- *Transposition*, is a personalized all-to-all primitive that carries out a transposition of messages on the processes as follows:

$$\begin{array}{cccc} P_0 & P_1 & P_2 & P_3 \\ M_{0,3} & M_{1,3} & M_{2,3} & M_{3,3} \\ M_{0,2} & M_{1,2} & M_{2,2} & M_{3,2} \\ M_{0,1} & M_{1,1} & M_{2,1} & M_{3,1} \\ M_{0,0} & M_{1,0} & M_{2,0} & M_{3,0} \end{array} \rightarrow \begin{array}{cccc} P_0 & P_1 & P_2 & P_3 \\ M_{3,0} & M_{3,1} & M_{3,2} & M_{3,3} \\ M_{2,0} & M_{2,1} & M_{2,2} & M_{2,3} \\ M_{1,0} & M_{1,1} & M_{1,2} & M_{1,3} \\ M_{0,0} & M_{0,1} & M_{0,2} & M_{0,3} \end{array}$$

We have $P^2$ messages $M_{r,k}$ with $P$ messages $M_{r,k}$ stored on $P_r$, and after the transposition, we have the $M_{r,k}$ stored on $P_k$ for all $0 \leq k \leq P-1$. This transposition primitive is very useful for inverting matrices partitioned in blocks on the grid or torus topology, for example.

- The *circular shift* operates a global shift of messages as follows:

$$M_0 \quad M_1 \quad M_2 \quad M_3 \rightarrow M_3 \quad M_0 \quad M_1 \quad M_2$$

The $P$ messages $M_k$ are stored locally and the message $M_{(k-1) \bmod P}$ is stored at $P_k$ in the output.

## 2.8 Communications on the Ring Topology with MPI

In Chap. 5, we shall consider distributed algorithms for the matrix multiplication on the ring and torus topologies. Here, we illustrate a short MPI program using the blocking communication primitives `send` and `receive` to perform a broadcast operation:

WWW source code: `MPIRingBroadcast.cpp`

```
// filename: MPIRingBroadcast.cpp
#include <mpi.h>
int main(int argc, char *argv[]) {
int rank, value, size;
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

do {
if (rank == 0) {scanf("%d", &value );
/* Master node sends out the value */
MPI_Send( &value, 1, MPI_INT, rank + 1, 0,
    MPI_COMM_WORLD);
}
else {
/* Slave nodes block on receive the send on the
    value */
MPI_Recv( &value, 1, MPI_INT, rank - 1, 0,
    MPI_COMM_WORLD, &status);
if (rank < size - 1) {
MPI_Send( &value, 1, MPI_INT, rank + 1, 0,
    MPI_COMM_WORLD);
}
printf("process %d got %d\n", rank, value);
} while (value >= 0);

MPI_Finalize();
return 0;
```

## 2.9 Examples of MPI Programs with Their Speed-Up Analysis

Let us now present different types of parallel implementations depending on the type of data transfer and on the local computations, and let us investigate the speed-up obtained on several problems. We recall that the speed-up is defined as follows:

$$s_p = \frac{t_1 = \text{time for one process}}{t_p = \text{time for } p \text{ processes}}.$$

We aim at reaching a *linear speed-up* in $O(P)$, where $P$ denotes the number of processes (each process ran to its own processor). In practice, one has to take care when accessing data (communication time, the different hierarchical levels of cache memories, etc.). In particular, we need to partition data when the data size are too big to hold on a single local memory of a processor (horizontal or vertical data partitioning).

Often, one can obtain a nice parallelization whenever the considered problem is said to be *decomposable*. For example, when playing chess, we need to find the best move given a configuration of the chessboard. Although the space of chess configuration is combinatorially very large, it is nevertheless finite, in $O(1)$. Thus in theory, one could explore all potential moves: at each move, we partition the space called the *configuration space*. The communication stages are for partitioning the problem into sub-problems and for combining the solutions of the sub-problems (`reduce`). Therefore, one expect to obtain a linear speed-up. In practice, the chess performance of a parallel software depends on the depth of the search tree it uses for exploring the configuration space. High Performance Computing has been instrumental in designing such a powerful chess software that won against Human: in 1997, Kasparov lost a chess play to the computer named Deeper Blue using 12 GFLOPS. Nowadays, people are focusing on the go game that offers a bigger combinatorial space. Advances in go program performance also implies progress in many other technical fields.

However, not all problems can be easily or well parallelized. For example, problems using *irregular* and *dynamic domains* like when simulating snow melting (that requires to dynamically and locally re-mesh domains, etc.). In that case, in order to obtain a good speed-up, we require to explicitly manage the load balancing among the processes. Splitting dynamically data among processes cost a lot since it requires to transfer data, and the overall speed-up is difficult to predict, because it depends on the semantic of the problem on the considered input data-sets, etc.

Let us know consider some very simple illustrative MPI programs.

### 2.9.1 The Matrix–Vector Product in MPI

The chapter on linear algebra (Chap. 5) will concentrate on distributed algorithms on the oriented ring and torus topologies.

WWW source code: `MPIMatrixVectorMultiplication.cpp`

```
// filename: MPIMatrixVectorMultiplication.cpp
#include <mpi.h>

int main(int argc, char *argv[]) {
  int A[4][4], b[4], c[4], line[4], temp[4],
      local_value, myid;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  if (myid == 0) {/* initialization */
    for (int i=0; i<4; i++) {
      b[i] = 4 ? i;
      for (int j=0; j<4; j++)
        A[i][j] = i + j;
    }
    line[0]=A[0][0];
    line[1]=A[0][1];
    line[2]=A[0][2];
    line[3]=A[0][3];
  }

  if (myid == 0) {
    for (int i=1; i<4; i++) {// slaves perform
        multiplications
      temp[0]=A[i][0];
      temp[1] = A[i][1];
      temp[2] = A[i][2];
      temp[3] = A[i][3];
      MPI_Send( temp, 4, MPI_INT, i, i,
          MPI_COMM_WORLD);
      MPI_Send( b, 4, MPI_INT, i, i, MPI_COMM_WORLD)
          ;
    }
  } else {
    MPI_Recv( line, 4, MPI_INT, 0, myid,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv( b, 4, MPI_INT, 0, myid, MPI_COMM_WORLD
        , MPI_STATUS_IGNORE);
  }
  {// master node
    c[myid] = line[0] * b[0] + line[1] * b[1] + line
        [2] * b[2] + line[3] * b[3];
    if (myid != 0) {
```

```
        MPI_Send(&c[myid], 1, MPI_INT, 0, myid,
            MPI_COMM_WORLD);
      } else {
        for (int i=1; i<4; i++) {
          MPI_Recv( &c[i], 1, MPI_INT, i, i,
              MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
      }
      MPI_Finalize();
      return 0;
    }
```

### 2.9.2 Example of MPI Reduce Operations: Computing the Factorial and Minimum Value of an Array

The following code illustrates how to perform a global calculation in MPI using a collective reduce operation:

WWW source code: `MPIFactorialReduce.cpp`

```
// filename: MPIFactorialReduce.cpp
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[]) {
  int i,me, nprocs;
  int number, globalFact=-1, localFact;

  MPI_Init(&argc,&argv);

  MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&me);

    number=me+1;
    MPI_Reduce(&number,&globalFact,1,MPI_INT,
        MPI_PROD,0,MPI_COMM_WORLD);
    if (me==0) printf("Computing the factorial in
        MPI: %d processus = %d\n",nprocs,globalFact);

    localFact=1; for(i=0;i<nprocs;i++) {localFact*=
        (i+1);}
    if (me==0) printf("Versus local factorial: %d\n"
        ,localFact);

MPI_Finalize();
}
```

We now turn to a more elaborate example: computing the global minimum value of a set of arrays stored in the local memories of processes:

WWW source code: `MPIMinimumReduce.cpp`

```
// filename: MPIMinimumReduce.cpp
#include <mpi.h>
#include <stdio.h>

#define  N   1000

int main(int argc, char** argv) {
  int rank, nprocs, n, i;
  const int root=0;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  float val[N];
  int myrank, minrank, minindex;
  float minval;

  // fill the array with random values (assume UNIX here)
  srand(2312+rank);
  for (i=0; i<N; i++) {val[i]=drand48();}

  // Declare a C structure
  struct {  float value;  int   index; } in, out;

  // First, find the minimum value locally
  in.value = val[0]; in.index = 0;
  for (i=1; i <N; i++)
    if (in.value > val[i]) {
      in.value = val[i];  in.index = i;
    }

  // and get the global rand index
  in.index = rank*N + in.index;

  // now the compute the global minimum
    // the keyword in MPI for the binary commutative operator
       is MPI_MINLOC
  MPI_Reduce( (void*) &in, (void*)  &out, 1,
      MPI_FLOAT_INT, MPI_MINLOC, root, MPI_COMM_WORLD
       );
```

```
    if (rank == root) {
      minval = out.value; minrank = out.index / N;
          minindex = out.index % N;
      printf("minimal value %f on proc. %d  at
          location %d\n", minval, minrank, minindex);
    }

    MPI_Finalize();
}
```

### 2.9.3 Approximating $\pi$ with Monte-Carlo Stochastic Integration

We describe the Monte-Carlo sampling approach to approximate a complex integral calculation by a discrete sum. Loosely speaking, Monte-Carlo sampling is bypassing the continuous integral $\int$ calculation by approximating it with a discrete sum: $\int \approx \sum$. To approximate $\pi$ (an irrational number), we draw randomly $n$ points uniformly inside the unit square. We then compute the ratio of the number of points $n_c$ falling inside the unit disk positive orthant over the total number of drawn points. Therefore we can deduce that:

$$\frac{\pi}{4} \approx \frac{n_c}{n}, \pi_n = \frac{4n_c}{n}$$

The approximated value of $\pi$ converges very slowly in practice, but this estimator is proven to be statistically consistent since we have the following theoretical result:

$$\lim_{n \to \infty} \pi_n = \pi.$$

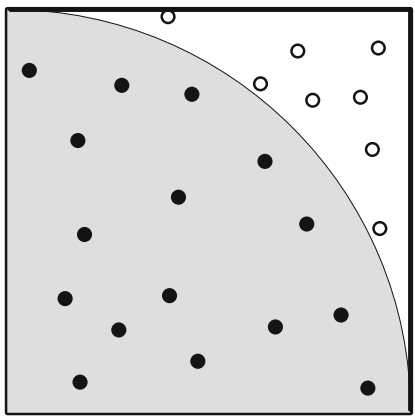

**Fig. 2.7** Monte-Carlo rejection sampling to approximate $\pi$: we draw at random $n$ points uniformly in the unit square, and we count the number of points that fall within the unit radius circle centered at the origin ($n_c$). We approximate $\frac{\pi}{4}$ as the ratio $\frac{n_c}{n}$

Moreover, this approach is pretty easy to parallelize, and the speed-up is linear, as expected. Figure 2.7 illustrates this Monte-Carlo stochastic estimation of $\pi$ by the method called *rejection sampling*.

WWW source code: `MPIMonteCarloPi.cpp`

```
// filename: MPIMonteCarloPi.cpp
int main(int argc, char *argv[]) {
  MPI_Init(&argc, &argv);
  #define INT_MAX_ 1000000000
    int myid, size, inside=0, outside=0, points
        =10000;
  double  x, y, Pi_comp, Pi_real
      =3.14159265358979323846264 3;

  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  if (myid == 0) {
    for (int i=1; i<size; i++) /* send  to  slaves
        */
      MPI_Send(&points, 1, MPI_INT, i, i,
          MPI_COMM_WORLD);
  } else
    MPI_Recv(&points, 1, MPI_INT, 0, i,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  rands=new double[2*points];

  for (int i=0; i<2*points; i++ ) {
    rands[i]=random();
    if (rands[i]<=INT_MAX_)
      i++
    }

    for (int i=0; i<points; i++ ) {
      x=rands[2*i]/INT_MAX_;
      y
        =rands[2*i+1]/INT_MAX_;
      if ((x*x+y*y)<1) inside++ /* point  inside
          unit circle*/
    }

  delete[] rands;

  if (myid == 0) {
    for (int i=1; i<size; i++) {
      int temp;
      MPI_Recv(&temp, 1, MPI_INT, i, i,
          MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      inside+=temp;
    } /* master sums all  */
  } else
```

```
    MPI_Send(&inside, 1, MPI_INT, 0, i,
        MPI_COMM_WORLD); /* send inside to master */
  if (myid == 0) {
    Pi_comp = 4 * (double) inside / (double)(size*
        points);
    cout << "Value obtained: " << Pi_comp << endl <<
         "Pi:" << Pi_real << endl;
  }

  MPI_Finalize();

  return 0;
}
```

### 2.9.4 Monte-Carlo Stochastic Integration for Approximating the Volume of a Molecule

A molecule $M$ is modeled by a set of $n$ 3D spheres where each sphere represents an atom (with given location and radius). We would like to compute the volume $v(M)$ of the molecule $M$ (that is, the volume of the union of spheres). We shall approximate this volume by performing a stochastic approximation: first, we compute an enclosing bounding box BB, and then we perform rejection sampling inside this bounding box. We draw a set of $e$ uniform variates inside BB and count the number of variates $e'$ falling inside the union of spheres. Then we approximate $v(M)$: $v(M) \simeq \frac{e'}{e} v(\text{BB})$.

Figure 2.8 illustrates the Monte-Carlo rejection sampling for computing the union of a set of 2D balls. The sequential code is given below:

WWW source code: `SequentialVolumeUnionSpheres.cpp`

```
// filename: SequentialVolumeUnionSpheres.cpp
// Sequential implementation of the approximation of the
    volume of a set of spheres
#include <limits>
#include <math.h>
#include <iostream>
#include <stdlib.h>
#include <time.h>

#define n 8*2
#define d 3
#define e 8*1000

double get_rand(double min, double max) {
    double x = rand() / (double)RAND_MAX;
    return x * (max - min) + min;
}
```

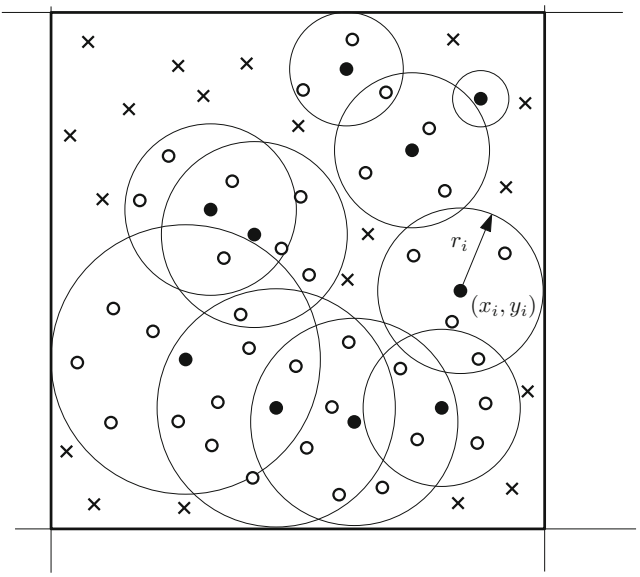


**Fig. 2.8** Monte-Carlo rejection sampling to approximate the volume of the union of balls $M$: first, we compute the bounding box (BB) of the balls. Then we draw uniformly inside BB $e$ samples and approximate the volume of the union of balls by $v(M) \simeq \frac{e'}{e} v(\text{BB})$

```
double distance2(double p0[d], double p1[d]) {
    double x = 0;
    for (int i = 0; i < d; i++) {
        double diff = p0[i] - p1[i];
        x += diff * diff;
    }
    return x;
}

int main(int argc, char** argv) {
    srand(0);
    double radius[n];
    double C[n][d];
    // Generate data
    for (int i = 0; i < n; i++) {

        radius[i] = get_rand(1, 5);
        for (int j = 0; j < d; j++) C[i][j] =
            get_rand(-20, 20);

    }
    // Compute bounding box
    double bb[d][2];
    for (int i = 0; i < d; i++) {
```

```
            bb[i][0] = std::numeric_limits<double>::
                infinity();
            bb[i][1] = -std::numeric_limits<double>::
                infinity();

        }
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < d; j++) {
                bb[j][0] = fmin(bb[j][0], C[i][j] -
                    radius[i]);
                bb[j][1] = fmax(bb[j][1], C[i][j] +
                    radius[i]);
            }
        }
        // Compute the volume of the bounding box
        double volBB = 1;
        for (int i = 0; i < d; i++) volBB *= bb[i][1] -
            bb[i][0];

        // Draw samples and perform rejection sampling
        int ePrime = 0;

        for (int i = 0; i < e; i++) {
            double pos[d];
            for (int j = 0; j < d; j++) pos[j] =
                get_rand(bb[j][0], bb[j][1]);
            for (int j = 0; j < n; j++) {

                if (distance2(pos, C[j]) < radius[j] *
                    radius[j]) {
                    ePrime++;
                }
            }
        }
        // Compute the volume
        double vol = volBB * (double)ePrime / double(e);
        std::cout << vol << std::endl;
        std::cout << ePrime << std::endl;
        return 0;
    }
```

Let us implement this sequential code on a cluster of machine. Initially, we assume that the collection of spheres is stored on the root process (say, of rank 0), and we shall distribute those data using a scattering operation. Then we compute in parallel the bounding box by taking the bounding box of local bounding boxes using reduce operations (with the binary operators `MPI_MIN` and `MPI_MAX`). Then the random variates are sampled by the root process and dispatched to all processes using another scattering operation. Then each process tests whether the variates fall inside the union of its local set of spheres, and finally we aggregate the accepted variates using another reduce operation with the logical OR as the binary operator: `MPI_LOR`.

The MPI implementation is given below:

WWW source code: `MPIVolumeUnionSpheres.cpp`

```
// filename: MPIVolumeUnionSpheres.cpp
// Parallel implementation of the approximation of the volume
    of a set of spheres
#include <limits>
#include <math.h>
#include <iostream>
#include <stdlib.h>
#include <time.h>
#include "mpi.h"

#define n 8*2
#define d 3
#define e 8*1000

double get_rand(double min, double max) {
    double x = rand() / (double)RAND_MAX;

}

double distance2(double p0[d], double p1[d]) {
    double x = 0;
    for (int i = 0; i < d; i++) {
        double diff = p0[i] - p1[i];
        x += diff * diff;
    }
    return x;
}

int main(int argc, char** argv) {
    srand(0);
    MPI_Init(&argc, &argv);

    int n_proc, rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &n_proc);


    double radius0[n];
    double C0[n][d];
    // Generate data
    if (rank == 0) {
        for (int i = 0; i < n; i++) {
            radius0[i] = get_rand(1, 5);
            for (int j = 0; j < d; j++) C0[i][j] =
                get_rand(-20, 20);
        }

    }
```

```
// Send data to processes
double radius[n];
double C[n][d];
int begin = n / n_proc*rank;
int loc_n = n / n_proc;
MPI_Scatter(radius0, loc_n, MPI_DOUBLE, &(radius
    [begin]), loc_n, MPI_DOUBLE, 0,
    MPI_COMM_WORLD);
MPI_Scatter(C0, 3 * loc_n, MPI_DOUBLE, &(C[begin
    ][0]), 3 * loc_n, MPI_DOUBLE, 0,
    MPI_COMM_WORLD);
double bb[d][2];

// Compute the bounding box
for (int i = 0; i < d; i++) {
     bb[i][0] = std::numeric_limits<double>::
         infinity();
     bb[i][1] = -std::numeric_limits<double>::
         infinity();

     for (int j = begin; j < begin + loc_n; j++)
         {
          bb[i][0] = fmin(bb[i][0], C[j][i] -
              radius[j]);
          bb[i][1] = fmax(bb[i][1], C[j][i] +
              radius[j]);
     }

     MPI_Reduce(rank ? &(bb[i][0]) : MPI_IN_PLACE
         , &(bb[i][0]), 1, MPI_DOUBLE, MPI_MIN, 0,
          MPI_COMM_WORLD);
     MPI_Reduce(rank ? &(bb[i][1]) : MPI_IN_PLACE
         , &(bb[i][1]), 1, MPI_DOUBLE, MPI_MAX, 0,
          MPI_COMM_WORLD);
}

// Compute the volume of the bounding box

double volBB = 1;
for (int i = 0; i < d; i++) volBB *= bb[i][1] -
    bb[i][0];
// Draw variates and perform rejection sampling
double samples[e][3];
if (rank == 0) {

     for (int i = 0; i < e; i++) {
          for (int j = 0; j < d; j++) samples[i][j
              ] = get_rand(bb[j][0], bb[j][1]);
     }
}
MPI_Bcast(samples, 3 * e, MPI_DOUBLE, 0,
    MPI_COMM_WORLD);
```

```
        // Testing variates

        bool hit[e];
        for (int i = 0; i < e; i++) hit[i] = false;
        for (int i = 0; i < e; i++) {
            for (int j = begin; j < begin + loc_n; j++)
                {
                 if (distance2(samples[i], C[j]) < radius
                    [j] * radius[j]) hit[i] = true;
            }
        }

        // Gather results and count the accepted variates

        bool hit0[e];
        for (int i = 0; i < e; i++) hit0[i] = false;

        MPI_Reduce(hit, hit0, e, MPI_C_BOOL, MPI_LOR, 0,
            MPI_COMM_WORLD);

        if (rank == 0) {
            int ePrime = 0;
            for (int i = 0; i < e; i++) {
                if (hit0[i]) ePrime++;
            }
            double vol = volBB * (double)ePrime / double
                (e);
            std::cout << vol << std::endl;
            std::cout << ePrime << std::endl;
        }

        MPI_Finalize();
        return 0;
    }
```

## 2.10 References and Notes

A precursor of the MPI standard was the software library *PVM*[18] that stands for *Parallel Virtual Machine*. That PVM library already had both the synchronous and asynchronous communication primitives. The MPI standard is well-covered in many textbooks dealing with *parallel computing*, see [3, 4] for example. In this chapter, we have only covered the main concepts and primitives of the MPI library. We recommend the interested reader these following books [5, 6] that fully cover all functionalities of the first and second standard versions (called MPI-I and MPI-II). There are many interesting mechanisms in the MPI standard that have been designed to facilitate parallel programming: for example, one can define derived types using

[18] http://www.csm.ornl.gov/pvm/.

`MPI_type_struct`, and so on. The last standard version is *MPI-3* and its usages are described in the recent book [7]. Parallel prefix operations (or scan) in MPI are well-studied, highly optimized and benchmarked in the research paper [8].

## 2.11 Summary

MPI is a standardized *Application Programming Interface* (*API*) that allows one to provide unambiguously the interface (that is, the declaration of functions, procedures, data-types, constants, etc.) with the precise semantic of communication protocols and global calculation routines, among others. Thus a parallel program using distributed memory can be implemented using *various* implementations of the MPI interface provided by several vendors (like the prominent OpenMPI, MPICH2,[19] etc.) Communications can either be *synchronous* or *asynchronous*, *bufferized* or not *bufferized*, and one can define *synchronization barriers* where all processes have to wait for each other before further carrying computations. There is a dozen MPI implementations available, and those implementations can further be called in many different languages (usually C, C++ and Python) using an appropriate *language binding* (a wrapper library to the underlying MPI implementation of the MPI standard). The last installment of the MPI standard is MPI-3 that it offers beyond the usual basic communication routines (broadcast, scatter, gather, all-to-all) over 200 functions that also allow one to also manage the Input/Output (I/O) in a distributed fashion as well.

## 2.12 Exercises

**Exercise 1** Consider the mathematical identity $\pi = \int_0^\infty \frac{4}{1+x^2} \mathrm{d}x$ to approximate $\pi$ by a Monte-Carlo stochastic integration. Fill the missing parts of the MPI program below (using the C++ binding):

WWW source code: `MPIPiApproximationHole.cpp`

```
// filename: MPIPiApproximationHole.cpp
#include <math.h>
#include "mpi.h"
#include <iostream>
using namespace std;

int main(int argc, char *argv[]){
    int n, rank, size, i;
    double PI  = 3.14159265358979323846264338;
    double mypi, pi, h, sum, x;
```

[19] http://www.mpich.org/.

```
    MPI::Init(argc, argv);
    size = MPI::COMM_WORLD.Get_size();
    rank = MPI::COMM_WORLD.Get_rank();

    while (1) {
        if (rank == 0) {
            cout << "Enter n (or an integer < 1 to
                exit) :" << endl;
            cin >> n;
        }

        MPI::COMM_WORLD.Bcast(...);
        if (n<1) {
            break;
        } else {
            h = 1.0 / (double) n;
            sum = 0.0;
            for (i = rank + 1; i <= n; i += size) {
                x = h * ((double)i - 0.5);
                sum += (4.0 / (1.0 + x*x));
            }
            mypi = h * sum;

            MPI::COMM_WORLD.Reduce(...);
            if (rank == 0){
                cout << "pi is approximated by " <<
                    pi
                      << ", the error is " << fabs(pi
                           - PI) << endl;
            }
        }
    }
    MPI::Finalize();
    return 0;
}
```

**Exercise 2** (*Monte-Carlo rejection sampling in MPI*) In statistics, to sample independently and identically variates following a probability density function $f(x)$ defined over a finite support $[m, M]$ with maximal mode $f_M$ (the largest value of $f(x)$), we can proceed as follows: sample a uniform variate $u_1$ from the uniform distribution $u_1 \sim U(m, M)$, and then sample a uniform variate $u_2$ from the uniform distribution $[0, F]$. Accept $u_1$ if $u_2 \leq f(u_1)$ and reject it otherwise. Intuitively speaking, to explain that this procedure produces independently and identically variates following density $f(x)$, consider throwing darts on a blackboard rectangle $[m, M] \times [0, F]$ and keep only the $x$-coordinate of darts falling below the density curve. The rejection sampling technique works for non-normalized densities $q(x)$ so that $f(x) = q(x)/Z$ where $Z = \int_x q(x)\mathrm{d}x$ is the implicit normalization factor (a constant). Implement a MPI procedure on $P$ processes that draws $n$ random variates of the truncated standard normal distribution defined on the support $[-1, 1]$, with unnormalized density

$q(x) = \exp(-\frac{x^2}{2})$. Observe that this is a generalization of the Monte-Carlo approximation method of $\pi$.

**Exercise 3** (*Computing the volume of the intersection of balls in MPI*) In Sect. 2.9.4, we provided some sequential and parallel implementations for approximating the volume of the union of balls by a Monte-Carlo stochastic approximation scheme. Show how to adapt the distributed MPI code for approximating the volume of the intersection of a set of balls.

## References

1. Kernighan, B.W., Ritchie, D.M.: The C Programming Language, 2nd edn. Prentice Hall Professional Technical Reference, Englewood Cliffs (1988)
2. Stroustrup, Bjarne: The C++ Programming Language, 3rd edn. Addison-Wesley Longman Publishing Co. Inc, Boston (2000)
3. Kumar, V., Grama, A., Gupta, A., Karypis, G.: Introduction to Parallel Computing: Design and Analysis of Algorithms. Benjamin-Cummings Publishing Co. Inc, Redwood City (1994)
4. Casanova, H., Legrand, A., Robert, Y.: Parallel Algorithms. Chapman and Hall/CRC numerical analysis and scientific computing. CRC Press (2009)
5. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI-The Complete Reference, Volume 1: The MPI Core, 2nd edn. MIT Press, Cambridge (1998). (revised)
6. Gropp, W.D., Huss-Lederman, S., Lumsdaine, A., Inc netLibrary: MPI: The Complete Reference. Vol. 2, The MPI-2 Extensions. Scientific and engineering computation series. MIT Press, Cambridge (1998)
7. Gropp, W., Hoefler, T., Thakur, R., Lusk, E.: Using Advanced MPI: Modern Features of the Message-Passing Interface. MIT Press (2014)
8. Sanders, P., Larsson Träff, J.: Parallel prefix (scan) algorithms for MPI. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface, pp. 49–57. Springer (2006)