An Introduction to Parallel Programming Using MPI

11

This chapter serves as an introduction to the *Message Passing Interface* (MPI), which is a widely used library of code for writing parallel programs on distributed memory architectures. It is not intended that you will learn much about parallel programming from reading this chapter—we would recommend that you use a dedicated textbook (such as those listed in the Further Reading section at the end of this book [9,10]) or tutorial if you wish to gain a detailed knowledge. However, this chapter should give you a *basic guide* to compiling and running parallel programs written using MPI. If you are likely to use a scientific library built on MPI (such as PETSc[1]) then what you learn here in this chapter should help to demystify some of the library calls, and enable you to begin to edit parallel programs written by other programmers.

11.1 Distributed Memory Architectures

There are several ways of classifying parallel computers and parallel programs but the most basic one is that of *shared memory* versus *distributed memory* machines. In the shared memory architecture several processing units (often called "cores" nowadays), share access to a common pool of memory, as shown in Fig. 11.1. This architecture has the advantage that a part of a program running on one core can easily communicate with another, since it can read or write in the memory space of the other part of the program. Programming for shared memory can be quite easy and the programs are generally very quick, but historically shared memory machines

[1]The Portable Extensible Toolkit for Scientific Computing (PETSc, pronounced "pet see") is a library providing functionality for the solution of linear and nonlinear systems of equations on both sequential and parallel architectures.

J. Pitt-Francis and J. Whiteley, *Guide to Scientific Computing in C++*, Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-319-73132-2_11

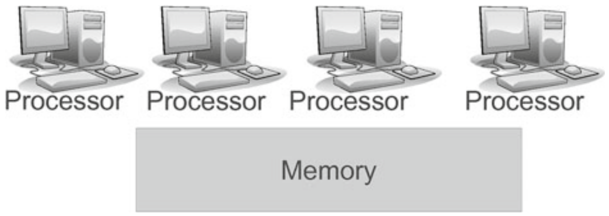

Fig. 11.1 A shared memory parallel architecture: the processors/cores are co-located and share a common memory

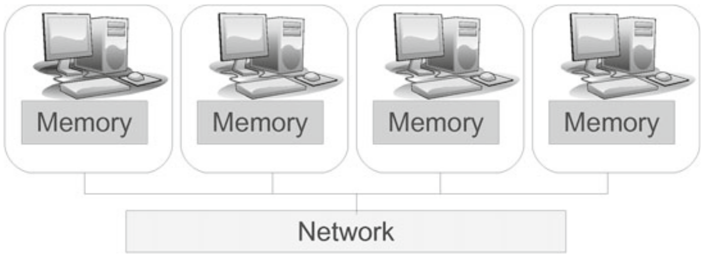

Fig. 11.2 A distributed memory parallel architecture: each processor has sole access to its local memory and the machines are connected on the same network

have been expensive, require specialised hardware, and physical constraints limit the total number of cores. This situation is, however, now changing as most new desktop computers have two or more cores.

The other main architecture commonly used for parallel programming is the distributed memory architecture (see Fig. 11.2), where each processing unit has a local memory space where it can read and write with ease, but the memory of other processors is completely hidden. The processors are connected—allowing data to be communicated between processors—on a network which could be a dedicated fast switch network (in the case of a cluster computer) or could be the wider Internet. The existence of the network between the processing units means that programming for this architecture is likely to be more complicated, and that programs that rely heavily on communication between processors are likely to be slower. However, as we shall now explain, distributed memory programs are versatile.

The versatility of distributed memory programs is evidenced by the fact that it is possible to take a program intended for a distributed memory architecture and run it on a shared memory architecture. In this case, the individual parts of the parallel program will have separate memory spaces within the shared memory system (so that they cannot directly access each others' memory), but will be able to communicate via the memory system. Communication is therefore much faster than over a network. Distributed memory programs can readily be executed on shared memory machines and are fast, but are also memory-hungry. The reverse is not true: you cannot, in general, run a shared memory program on a distributed memory cluster.[2]

You can even run distributed memory programs on a computer with a single processor. All the parallel processes will be run as what are known as individual *threads*, and will communicate via the memory system with the operating system responsible for context switching between the threads. There is no performance advantage to doing this, since there is an overhead to run many threads on a single processor. The advantage is that you can write and debug a program on a low-powered laptop, tune it on a shared memory desktop, and then deploy exactly the same program on a supercomputer.

11.2 Installing MPI

MPI is actually a set of standards for performing distributed computing. The MPI-1 standard documents the primary core of MPI (basic point-to-point and collective communication) while the MPI-2 standard adds other useful but advanced features such as parallel file access (through the input and output operations provided by MPIIO) and remote memory access (one-sided communication). Because MPI is a set of open standards there are various implementations available to choose from. The most commonly used are the MPICH and Open MPI implementations. The current versions of MPICH and Open MPI (formerly LAM/MPI) both implement all the functionality in both the MPI-1 and MPI-2 standards.

Both MPICH and Open MPI are open source projects, under active development and freely available to download. They may be run on a wide variety of machine architectures, operating systems and communication infrastructures. The Open MPI library implementation is currently available from major Linux distribution repositories and is therefore easy to install on Linux systems. It is configured so that it can be used either on a stand-alone system (in the manner of a shared-memory system) or across standard Ethernet using the secure shell `ssh` protocol.

11.3 A First Program Using MPI

Just as in Sect. 1.2, we introduce the MPI library by using a program that prints the text “Hello World” to the screen. This time, it runs and prints in parallel. This simple example C++ MPI program is shown below.

[2]There are several programming libraries which allow the programmer access to a *distributed shared memory* computer where machines over a network act as if they were part on one contiguous system. There has, however, not been wide-spread use of these libraries at the time of writing.

Before explaining the purpose of the individual statements in this program, we need to explain what we mean by the term *process*. Loosely speaking a process is a part of a parallel program that may be executed independently of the other parts, provided that data can be communicated through MPI calls when required. As such, a process can be thought of as a component of the program that can be executed on one of the processors shown in Fig. 11.2. (However, we make a distinction between processes and physical processors—or cores—because it is possible to run multiple processes on a single processor.) If a code has p processes, then each process is given a `rank` which is a unique integer in the range $0 \leq \text{rank} < p$.

Listing 11.1 `MpiHelloWorld.cpp`

```
 1  #include <iostream>
 2  #include <mpi.h>
 3
 4  int main(int argc, char* argv[])
 5  {
 6     MPI::Init(argc, argv);
 7
 8     int num_procs = MPI::COMM_WORLD.Get_size();
 9     int rank = MPI::COMM_WORLD.Get_rank();
10     std::cout << "Hello world from process " << rank
11               << " of " << num_procs << "\n";
12     MPI::Finalize();
13     return 0;
14  }
```

There are several lines of the program above which mention MPI. The first of these is the extra include on line 2 which allows the program to see the full functionality of the MPI library. Subsequently, there are `MPI::Init` and `MPI::Finalize` statements on lines 6 and 12 which start and stop the parallel part of the code. All MPI calls must lie between these two statements. The method `Get_size` allows us to access the number of processes taking part in the program execution, and the method `Get_rank` allows us to identify the process which is executing a given statement. The `COMM_WORLD` object represents a communications group involving *all the processes* running the current calculation. It is possible to split this communication group up into smaller groups so that subsets of the processes can share private data.

It should be noted that all the calls to MPI in program above use calls to specific C++ bindings to the MPI library. So `Finalize` is a function in the MPI *namespace* (see Sect. B.4) and `Get_size` is a method of the communication object `COMM_WORLD`. Some C++ programmers prefer not to use these bindings, but opt instead for the plain C functions `MPI_Init`, `MPI_Get_size`, etc. which have a slightly different syntax. Both versions are valid in C++ programs and can even be mixed.

11.3.1 Essential MPI Functions

The functions `MPI::Init` and `MPI::Finalize` on lines 6 and 12 of Listing 11.1 are required calls in any MPI program. On line 6, `MPI::Init` is able to promote the program from a single executable to a parallel program running as several processes. In order to do this, it needs to know how many processes to launch and on which machines they should be run—as we shall see in Sect. 11.3.2 this is information that can be made available via command-line arguments. `MPI::Init` inspects the command-line arguments provided by `argc` and `argv`, acting on any it recognises.

Some MPI implementations of `MPI::Init` update their arguments by removing those which have been acted upon. Therefore, if you want an MPI program to read some specific arguments from the command-line for use in your calculation, the best place to do this is *after* `MPI::Init` since, at that point, all MPI-specific arguments have been read and updated as necessary.

The `MPI::Finalize` makes sure that the program closes down neatly, closing any remote connections and terminating all processes.

11.3.2 Compiling and Running MPI Code

So far when we have compiled C++ programs we have included code either from standard locations (through including files such as `cmath`) or from other parts of our own code (such as `Book.hpp`). MPI, as a *third-party library*, is not part of the standard C++ distribution.

Normally when you compile against a third-party library, you would have to include extra compiler flags specifying the location of the header files, the location of the libraries themselves, and names of some of the library dependencies. This can be a little onerous. Added to this, on some large computing facilities there may be several versions of MPI available which make it possible to accidentally compile some of your program with one version, and the remainder of the program with another, possibly incompatible, version. To ameliorate these difficulties, the MPI distributions have provided "wrapper compilers" for C++ (as well as for C and Fortran). The wrapper compiler automatically adds the correct compiler flags when it calls the actual compiler. The C++ MPI compiler on most systems will be called `mpiCC`, `mpic++` or `mpicxx`. It is probably the case that it exists with more than one synonym.

The standard Linux distribution of the Open MPI package has an `mpiCC` compiler which is a wrapper to the GNU `g++` C++ compiler. To ensure that this compiler is installed, open a terminal window and type "`which mpiCC`" followed by return. Hopefully the computer will respond by reporting the location of this compiler, for example,

```
$ which mpiCC
/usr/bin/mpiCC
$
```

To compile the code given in Listing 11.1, open a terminal window and create a directory where code may be saved. Move into this directory, and save the code as "`MpiHelloWorld.cpp`". The MPI wrapper compiler may have some compiler flags of its own, but most flags are passed on to the normal `g++` compiler. In the same directory type,

```
mpiCC -o MpiHelloWorld MpiHelloWorld.cpp
```

It is possible (but uninteresting) to run the executable which you have just produced as a standalone program. That is, without any of the MPI machinery and with no code run in parallel. If this is the case then `Get_size` will return 1, and, because there is only one process, `Get_rank` on that process will return 0. Just as in array numbering, the process rank numbering starts at zero, so each process is given a unique integer rank in the range $0 \le \text{rank} < p$, where p is the total number of processes.

```
$ ./MpiHelloWorld
Hello world from process 0 of 1
```

To run in parallel either on the same machine or across a network or cluster, use the `mpirun` command (also known as `mpiexec` on some MPI implementations). This command will, if necessary, launch a service (called a daemon) on all the machines involved in the calculation. It will then make sure that copies of the executable can be run on every machine.

To run the program locally, use the "number of processes" `-np` flag.

```
$ mpirun -np 2 ./MpiHelloWorld
Hello world from process 0 of 2
Hello world from process 1 of 2
```

To run the program across a network you can give a list of machines in a *host file*, or alternatively list the machine names on the command-line. It is imperative that you have an account on the remote machines, that you are able to connect via `ssh` (preferably without being prompted for a password), that the machines have the same MPI implementation installed on them, and that they are capable of running the executable file which you are sending. In the example below, ranks 0 and 2 of a 3-process job are launched on remote machines. The rank 1 process will run on the local machine, from where the job has been launched. Note that in this case buffered output from the local machine has appeared on the screen before output which has been sent from the remote machines.[3]

```
$ mpirun -host remote1.org,localhost,remote2.org ./MpiHelloWorld
Hello world from process 1 of 3
Hello world from process 2 of 3
Hello world from process 0 of 3
```

If you are running your program on a large cluster or a supercomputer, then it is likely that the program will be launched from a script via a queueing system. In this case, the locations of the processors available to you will be determined by the job queue manager. You should obtain detailed instructions from the system administrators about which arguments to give to the `mpirun` command in your script.

11.4 Basic MPI Communication

While the parallel "Hello World" program used the MPI libraries, it did not make any use of the communication features offered by these libraries. More specifically, it did not do any *message passing*, which is the main feature of MPI. In this section, we give a brief survey of some of the common communication patterns available in the MPI library through providing a sample of the large range of available function calls.

11.4.1 Point-to-Point Communication

The essential part of MPI functionality is being able to send a single message between processes, where one process *sends* while another process *receives*. These two functions are called `Send` and `Recv`. Their function prototypes are:

```
void Comm::Send(const void* buf, int count,
                const Datatype& datatype,
                int dest, int tag) const
```

[3]MPI implementations vary in how they return console input from the individual processes to the console from which the program was launched. Even when `flush` is called on the `cout` stream it may still be the case that the MPI machinery is buffering output.

```
void Comm::Recv(void* buf, int count,
                const Datatype& datatype,
                int source, int tag) const
```

The `Send` method takes data on the current process from the location given by the pointer `buf`. These data are assumed to be in contiguous memory (as an array of `count` variables), but `buf` may be a pointer to a single variable. Note the `const` keyword next to the buffer argument: MPI is making a guarantee not to alter your data during the message sending. The `datatype` field tells the system what the type of the data is (so that the correct number of bytes are sent in the correct format). The last two arguments of the `Send` method give the destination process number (this is the *rank* of the process we wish to send to) and a *tag*. The message tag can be any nonnegative integer value and its purpose is to allow the user to easily identify the context of a message. Negative tag values are reserved by the library for special values such as `MPI::ANY_TAG` which is introduced below.

The `Recv` method has the same basic arguments: a pointer to a buffer in which to store the message, an integer `count` that gives the expected number of items in the message, the data-type for these items, the rank of the source process which is sending the message and the tag value of a message. The receiver is allowed to use wild-cards for either the source of the message, or the message tag, or both. The wild-card `MPI::ANY_SOURCE`[4] is useful if, for example, we wish to receive all the results of one phase of computation (tagged with `phase_1_tag`, for example) before moving on to the next phase. Messages sent with the next tag (`phase_2_tag`) can then be queued until the receiving process is ready for them. The wild-card `MPI::ANY_TAG` is useful if we know which process is sending the data, but do not know what the tag will be.

The corresponding MPI `Datatype` signatures for the types introduced in Chap. 1 are `MPI::BOOL`, `MPI::CHAR`, `MPI::INT` and `MPI::DOUBLE`.[5] There is no MPI type for strings because `std::string` is a C++ class rather than a plain data-type. It is possible to send entire C++ classes in MPI messages by using advanced programming features to introduce user-defined data-types, but this is not recommended. Classes can readily be transferred by packing the raw data into a message at one end and unpacking it into a waiting class at the other end.

The following code fragment illustrates sending one message consisting of two floating-point numbers from process 0 to process 1. Note that code involving point-to-point communication is necessarily nonsymmetric: both processes are running exactly the same program with the same code, but parts of the program which are intended only for one process are placed in specific blocks guarded by their process rank.

[4] `MPI::ANY_TAG` and `MPI::ANY_SOURCE` are C++ names for these wild-card values. Many codes use the interchangeable C names: `MPI_ANY_TAG` and `MPI_ANY_SOURCE`.

[5] Note that these are the C++ object names for these types—they are also called synonymously by their C names: `MPI_BOOL`, `MPI_CHAR`, `MPI_INT` and `MPI_DOUBLE`.

Listing 11.2 Example code for sending and receiving using the MPI libraries

```
int tag = 30;
if (MPI::COMM_WORLD.Get_rank() == 0)
{
   //Specific send code for process 0
   double send_buffer[2] = {100.0, 200.0};
   MPI::COMM_WORLD.Send(send_buffer, 2,
                        MPI::DOUBLE, 1, tag);
}
if (MPI::COMM_WORLD.Get_rank() == 1)
{
   //Specific receive code for process 1
   double recv_buffer[2] = {0.0, 0.0};
   MPI::COMM_WORLD.Recv(recv_buffer, 2, MPI::DOUBLE,
                        MPI::ANY_SOURCE, MPI::ANY_TAG);
   std::cout << recv_buffer[0] << "\n";
   std::cout << recv_buffer[1] << "\n";
}
```

11.4.1.1 Blocking and Buffered Sends

The default means of sending point-to-point messages with `Send` and `Recv` represents one combination in a spectrum of available communication protocols. Both functions are known as *blocking* functions because they do not allow the execution of the program to continue until it is safe to do so. The `Send` method not only guarantees that it will not change the contents of the data buffer, but that any subsequent changes to the data buffer will not affect the message that is being sent. So if computation is allowed to proceed from a `Send` call it either means that the message has already been delivered or that the data has been copied into another buffer ready for delivery.

The default `Send` is a compromise between the safety of waiting to be sure that a message has been delivered and the efficiency of getting on with other tasks after sending the message immediately. The other send functions have similar function prototypes, but slightly different names. We briefly describe these send functions below: the interested reader should consult a dedicated MPI programming book (such as [9,10]) for more details.

- The very safest, but possibly most inefficient, means of sending a message is to use a *blocking* synchronous send, `Ssend`. This function guarantees not to continue until the message has been delivered. This is a little like delivering a message by telephone conversation, because we cannot get on with our lives until the call has been made and the information has been relayed.
- A slightly more configurable version is `Bsend`, the *buffered* send. Like the plain `Send`, it allows the program to continue when safe, but this may happen faster since the message is copied to a separate buffer. This buffer must be supplied and configured by the user.

- At the top end of the spectrum, the most efficient means of sending a message is the *immediate* send `Isend`, which returns control to the program immediately, whether the message has been delivered, buffered or not yet acted on. This is a little like communicating via SMS text message in which we are able to press "send" and get on with other things safe in the knowledge that the recipient will get the information some time soon. Because it may be dangerous to overwrite the original data contained in the message, MPI provides functions for testing whether or not the message has been delivered. The `Isend` command gives back a handle (called an `MPI::Request`) which has a `Wait` method: this method instructs execution to "wait here" until the message has been sent.
- There are a few other flavours of `Send` including compatible combinations: an immediate send can make use of a user-supplied buffer using the buffered non-blocking combination `Ibsend`.

The default `Recv` function is also one of a spectrum of functions. It is technically a *blocking* function, because execution cannot continue until a suitable message has been received. There is also a *non blocking* immediate receive `Irecv` together with some utilities for probing whether there are any queued messages which match certain sources or tags. This means that your program, rather than waiting for messages to be received, could get on with useful work, occasionally going back to check for new information.

11.4.2 Collective Communication

Code for point-to-point communication is not symmetric: one process sends while another receives. MPI provides specialised collective calls in which all the processes take part by executing the same commands. There are several major different flavours of these communication patterns: the combined send-receive (where every process sends a message to another, while also receiving a remote message); one-to-many operations such as *broadcast* where data from one process are sent to the entire group; and many-to-one operations such as *reduction* in which an operation is used to combine results from all processes into a single result.

These collective calls have the advantage that they can be highly tuned in an MPI implementation to fit the local architecture. The broadcast of a single number to all p processes from process 0 could be achieved by sending $p - 1$ messages from process 0, one message to each of the recipients. However, if process 0 sends to only two processes who each send to two more, then the information is broadcast to all recipients in about $\log_2 p$ rounds of message sending. If a supercomputer consists of several multicore computers connected by Ethernet, then the broadcast algorithm can be tuned to minimise the number of Ethernet messages while possibly increasing the number of faster messages between cores in the same machine.

11.4.2.1 Barrier

The simplest collective method is `Barrier`. It says that every process should wait here until all processes are ready to proceed. Barriers are useful when you are timing certain parts of the code, printing out information to the console, or debugging the code. In the following code example there is a `Barrier` at line 3, the purpose of which is to ensure that all processes have completed writing their output (from line 1) before they are permitted to proceed.

```
1   std::cout << "Processes may arrive at any time\n";
2   std::cout.flush();
3   MPI::COMM_WORLD.Barrier();
4   std::cout << "All processes continue together\n";
5   std::cout.flush();
```

11.4.2.2 Combined Send and Receive

There are many cases when we might wish to send and receive many point-to-point messages at the same point in a computation, and where every process should be involved.

For example, consider solving a partial differential equation (PDE) using a finite difference scheme over a regular grid (such a grid is illustrated in two dimensions in Fig. 12.2) where the value of a variable at one position on the grid depends on the value of that variable at a few neighbouring grid points. A similar example from a different field is that of image processing: many image processing filters, such as edge detection or blurring, are implemented as weighted averages of image intensities over a small patch of neighbouring pixels. Such problems may be readily parallelised by dividing the grid or image into a number of identical vertical (or horizontal) strips and assigning one strip to each parallel process. Each process can compute its partition independently, except at the edges where information at grid-points or pixels assigned to the neighbouring process is required. A way to provide this information is to keep a local copy of the required neighbouring grid-point data and to update these data from the neighbouring process by message-passing. The local copy of remote neighbouring data is called *halo data* and the message passing process is called *halo exchange*.

Halo exchange is demonstrated in Fig. 11.3 using the example of an image processing filter. This filter produces an image where the image intensity in the processed image at a given pixel is the average of the image intensities in the original image at five pixels: the pixel of interest; the pixel to the left; the pixel to the right; the pixel directly above; and the pixel directly below. The pixels allocated to process `n` are those in the shaded area in Fig. 11.3. We may calculate the processed image at the pixels represented by open circles in this shaded region using only pixel intensities stored by process `n`. Before we may calculate the filtered image at the pixels represented by solid circles on the left edge of process `n`, however, we require access to the pixels represented by solid circles on the right edge of the pixels stored by

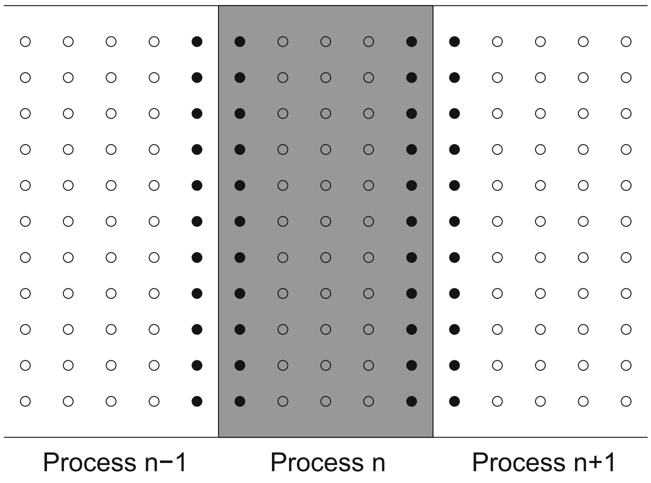

Fig. 11.3 Halo exchange between processes

process `n-1`: these pixels are referred to as the halo, and we need to copy these to process `n` before we can calculate the whole processed image. Similarly, the nodes along the left-hand boundary of process `n` must be copied to process `n-1` before the processed image may be calculated. This procedure of sending edge data in both directions between processes `n` and `n-1` is known as halo exchange. For the same reasons, two-way halo exchange is also required between processes `n` and `n+1`.

The partitioning of data between processes should ideally minimise the amount of data that has to be passed in halo exchanges: this is important when fine-tuning your code to produce optimum efficiency, but is beyond the scope of this book.

For these types of problem, a more sophisticated version of point-to-point message passing is the combined send and receive, called `Sendrecv`. Its function prototype is:

```
void Comm::Sendrecv(const void *sendbuf, int sendcount,
                    const Datatype& sendtype,
                    int dest, int sendtag,
                    void *recvbuf, int recvcount,
                    const Datatype& recvtype,
                    int source, int recvtag) const
```

Note that the ten arguments are divided into two sets of five: a set of send arguments about the outgoing message and a set of receive arguments about the incoming message. These are similar to the arguments given to the point-to-point versions in Sect. 11.4.1 and they are interpreted relative to the *local process*: if each process is sending to the rank above, by symmetry, each must be receiving from the rank below. It is possible to mix the types of messages (both in terms of `DataType` and

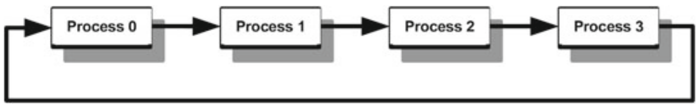

Fig. 11.4 Message passing between processes in a ring using combined send-receive

the length of the messages) so that, for example, odd-ranked processes are sending integer messages to the process above, but even-ranked processes are sending double precision floating point data. In this circumstance, on any given process the types of send and receive data will differ. As with the `Recv` functions, we can use the wild-cards for the source process identity and the received message tag.

The following code shows all processes communicating in a ring. Each process (with rank given by the variable `rank`) sends a message to its right-hand neighbour (`rank` + 1). Modular arithmetic—see Sect. 1.4.3—ensures that the `left_rank` and `right_rank` variables are set inside the range $0 \leq \text{rank} < \text{num\_procs}$ so that the top-most process is able to send a message to the rank 0 process. This message passing is illustrated schematically in Fig. 11.4 for four processes: the arrow indicates the direction in which the message is passed.

```
int tag = 30;
int rank = MPI::COMM_WORLD.Get_rank();
int num_procs = MPI::COMM_WORLD.Get_size();
// left_rank is rank-1
// Note modular arithmetic, so that 0 has
// neighbour num_procs-1
int left_rank = (rank-1+num_procs)%num_procs;
int right_rank = (rank+1)%num_procs;
int recv_data;
// Communicate in a ring ...->0->1->2...
MPI::COMM_WORLD.Sendrecv(&rank, 1, MPI::INT,
                         right_rank, tag,
                         &recv_data, 1, MPI::INT,
                         left_rank, tag);
std::cout << "Process " << rank << " received from "
          << recv_data << "\n";
```

There are cases, such as the halo exchange situation outlined above, where *nearly every* process will send halo data from the right-hand edge of its domain up to the next process to become a left-hand halo, but the top-most process does not need to send any data and the bottom-most process needs no left-edge halos. This is illustrated in Fig. 11.5 where four processes are taking part in the communication with the arrows indicating the direction in which information is passed. In a separate send-receive event the left-edges would also be sent down the chain to become right-edge halos, but again there is no need to send data from the bottom-most process. For this reason, MPI provides a special process name `MPI::PROC_NULL` which means that this process does not participate with a send and/or receive. This process name

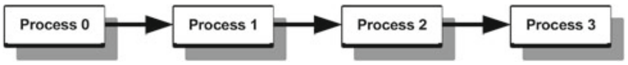

Fig. 11.5 Message passing between processes in a chain using combined send-receive. On process 3 the message destination is set to PROC_NULL

is illustrated in the following code, which is similar to the previous Sendrecv example, except that there is no closed loop: the top-most process does not send to process 0.

```
int tag = 30;
int rank = MPI::COMM_WORLD.Get_rank();
int num_procs = MPI::COMM_WORLD.Get_size();
int right_rank = rank+1;
// Top-most sends nowhere
if (rank == num_procs - 1)
{
   right_rank = MPI::PROC_NULL;
}
int left_rank = rank-1;
// Bottom-most receives nothing
if (rank == 0)
{
   left_rank = MPI::PROC_NULL;
}
int recv_data = 999; //This will be unchanged on proc 0
// Communicate 0->1->2... Final process sends nowhere
MPI::COMM_WORLD.Sendrecv(&rank, 1, MPI::INT,
                         right_rank, tag,
                         &recv_data, 1, MPI::INT,
                         left_rank, MPI::ANY_TAG);
std::cout << "Process " << rank << " received from "
          << recv_data << "\n";
```

11.4.2.3 Broadcast and Reduce

The collective operations broadcast and reduce are primarily one-to-many and many-to-one operations. In a broadcast (Bcast) operation, data from one process are shared with all other processes in the communication group. In a reduction operation all the data is concentrated to a single process. This reduction operation is likely to be of one of a standard set available for numerical data (MPI::MAX, MPI::MIN, MPI::SUM, and MPI::PROD). There are other predefined reduction operations available including some bit-wise operations, and there is also opportunity to define extra operations. The prototype signatures of the broadcast and reduce operations are given below. Note that the argument root is the *source* of the broadcast but the *destination* of the reduction. MPI also provides a many-to-many reduction operation Allreduce which may be thought of as a reduction operation followed by a broadcast to all processes.

```
void Comm::Bcast(void* buffer, int count,
                 const MPI::Datatype& datatype,
                 int root) const

void Comm::Reduce(const void* sendbuf, void* recvbuf,
                  int count, const MPI::Datatype& datatype,
                  const MPI::Op& op, int root) const
```

An example reduction operation is given in Sect. 11.5.1 where the partial sums of a series are summed together in a single reduction step. For now, here is a broadcast example in which one process—process 0—mimics throwing three dice by generating integer random numbers from 1–6 inclusive, and broadcasts the results of all three throws. Each process then adds their own rank to the value shown on the first die, and a reduction operation reports on the maximum value attained after this operation.

```
int dice[3] = {0, 0, 0};
//Proc 0 sets the dice (#sides)
if (MPI::COMM_WORLD.Get_rank() == 0)
{
   for (int i=0; i<3; i++)
   {
      dice[i] = (rand()%6)+1;
   }
}
//Proc 0 broadcasts
MPI::COMM_WORLD.Bcast(dice, 3, MPI::INT, 0);
//Every process adds their rank to dice[0]
dice[0] += MPI::COMM_WORLD.Get_rank();
//Reduce the first value to get the maximum
int max;
MPI::COMM_WORLD.Reduce(dice, &max, 1,
                       MPI::INT, MPI::MAX, 0);
//On Proc 0: max = dice[0]+MPI::COMM_WORLD.Get_size()-1
```

11.4.2.4 Scatter and Gather

The scatter and gather operations are extensions to broadcast and reduction operations. They are the most advanced operations which we cover in this book, and we do so because the gather operation is useful for taking data which has been distributed across processes and *concentrating* it onto a single process. For example, if a vector is split across processes in a similar manner to a PETSc vector we might wish to write it to a file using a single write operation using only one process.[6]

[6]There are a few standard ways of getting data to file from a parallel program: *concentration*, where one process does all the writing, as suggested above; *round-robin* where processes take it in turns to

The scatter operation Scatter is similar to the broadcast operation in that it is one-to-many with one process being responsible for sending the message to all other processors. Unlike the broadcast operation, where the same entries of data (of size count) are sent to all processes, the first count elements are send to the first process, the next count to the next and so on. MPI also provides a scatter for variable sized data (where the count size can be different for different destinations) which is called Scatterv.

The gather operation is similar to the reduce operation in that it is many-to-one with each process contributing some data to the result. The difference is that the data is not reduced but rather it is *concatenated*. If each process contributes count elements of data, then the gathering process must have space to store count multiplied by num_procs elements. There is a variable-sized data version of the gather, Gatherv in which the numbers of elements contributed per process may be different. MPI also provides Allgather and Allgatherv in which the result of the gather ends up on all the processes involved in the communication. These may be thought of as a regular Gather or Gatherv operation followed by a broadcast.

Below are the prototype signatures of the scatter and gather operations. For completeness we also give the signature of Allgatherv since we will demonstrate the use of Allgather and Allgatherv in Sect. 11.5.2.

```
void Comm::Scatter(const void* sendbuf, int sendcount,
            const MPI::Datatype& sendtype, void* recvbuf,
            int recvcount, const MPI::Datatype& recvtype,
            int root) const

void Comm::Gather(const void* sendbuf, int sendcount,
            const MPI::Datatype& sendtype, void* recvbuf,
            int recvcount, const MPI::Datatype& recvtype,
            int root) const

void Comm::Allgatherv(const void* sendbuf, int sendcount,
            const MPI::Datatype& sendtype, void* recvbuf,
            const int recvcounts[], const int displs[],
            const MPI::Datatype& recvtype) const
```

Most of the arguments in the above methods should be readily understood, since they are similar to the arguments of the previous less advanced methods. The argument root always refers the scatterer (sender) or to the gatherer (receiver). In most cases, the *types* and *counts* of the send and receive data should be identical, with the counts referring to the size of the array sent to (or received from) each process. In the variable size gather, the int array arguments recvcounts and displs are used to communicate the variable data counts and displacements for each process (so

open and close the same file; *parallel file libraries* such as MPI's MPIIO; and *separate files* where each process writes data to different places to be re-assembled later. The choice of output method is largely dependent on the data structure and size.

`recvcounts[rank]` should be equal `sendcount` for that process). The value `displs[rank]` contains the index in the gathered array `recvbuf` where the data from process `rank` should begin. There is some redundancy between the counts and displacements since one might expect the displacement of each process' data to be equal to the sum of the counts of the data from lower ranked processes. However, this redundancy allows there to be gaps in the gathered data.

11.5 Example MPI Applications

In this section, we give two examples of parallel programs written with MPI. The designs of the parallel algorithms shown here are not unique to the problems which they solve. In general, the choice of parallel algorithm depends on how it relates to an equivalent sequential algorithm (if there is one) and how the data is partitioned. One usually seeks to partition the data and computation between the processes in such a way that communication between processes is minimised and that the processes are given an equivalent amount of computational work. The task of giving the processes the same amount of work is known as *load balancing*.

However, merely giving each process a similar amount of work is no guarantee of a successful parallel algorithm if the combined computational load of parallel processes is much more than that of the sequential program, or if communication dominates the program. The measures of success in producing parallel programs are *parallel speedup* and *parallel efficiency*. The *parallel speedup* is the ratio of the time it takes to run the code sequentially on a certain problem to time it takes to run on p processes ($S_p = \frac{T_1}{T_p}$). In an ideal case, a problem can be partitioned such that it is well load balanced with minimal extra overhead, so we expect $S_p \simeq p$. Parallel efficiency scales this value by p: $E_p = \frac{T_1}{pT_p}$ so that E_p is generally in the range from 0 to 1 with 1 being the ideal value. It is uncommon, but not unusual, for a particular problem to scale in parallel such that $E_p > 1$. This fortunate situation normally arises when a given problem has memory constraints when run on a small numbers of processes and it is known as *super-linear speedup*.

11.5.1 Summation of Series

The summation of a series can be taken as an *abstraction* of a range of problems in which it is moderately easy to partition work between processes and there is minimal communication. Such problems are termed *embarrassingly parallel*. In the following example, the calculation is trivial but this case is representative of tasks which are possibly more labour intensive, such as Monte Carlo integration (see Exercise 11.4).

Consider the problem of summing a series, such as the approximation to π

$$\frac{\pi}{4} = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1},$$

credited to Gottfried Wilhelm Leibniz. Given that we cannot compute the sum to infinity, we approximate this summation with a finite sum from $n = 0$ to $n = max - 1$ for some value max (which may be assumed to be divisible by the number of processes, p). In dividing the max contributions between the processes evenly, we might choose to allocate this work in blocks, so that the first max/p contributions to the series go to process zero, and so on, or we might distribute in such a way as to interleave processor contributions. In the following example, the contributions are interleaved. Note that the only parallel communication needed in this code is a reduction operation, which combines the subtotals from the processes into a grand total for the entire calculation on process 0.

```cpp
#include <mpi.h>
#include <cmath>
#include <iostream>

//Program to sum Pi using Leibniz formula:
// Pi  = 4 * Sum_n ( (-1)**n/(2*n+1) )
int main(int argc, char* argv[])
{
   int max_n = 1000;
   double sum = 0;
   MPI::Init(argc, argv);

   int num_procs = MPI::COMM_WORLD.Get_size();
   int rank = MPI::COMM_WORLD.Get_rank();

   for (int n=rank; n<max_n; n+=num_procs)
   {
      double temp = 1.0/(2.0*((double)(n))+1.0);
      if (n%2 == 0) // n is even
      {
         sum += temp;
      }
      else
      {
         // n is odd
         sum -= temp;
      }
   }

   double global_sum;
   MPI::COMM_WORLD.Reduce(&sum, &global_sum, 1,
              MPI::DOUBLE, MPI::SUM, 0);
   if (rank == 0)
   {
      std::cout << "Pi is about " << 4.0*global_sum
                << " with error " << 4.0*global_sum-M_PI
                << "\n";
   }
   MPI::Finalize();
   return 0;
}
```

11.5.2 Parallel Linear Algebra

In this section, we give an outline of the operations required for performing parallel linear algebra operations. It is beyond the scope of this book to provide a complete parallel linear algebra library, but we outline some of the issues arising when we design such a system. A fundamental question to ask is how matrices and vectors might be partitioned across the processes. We choose to use the matrix-row partitioning (which will be described later) favoured by the PETSc library—although other parallel linear algebra systems, such as Mondriaan, use more sophisticated techniques.

We begin by discussing parallel implementation of the product between a matrix and a vector of suitable sizes. Using the matrix-row partitioning scheme, the matrix-vector product $\mathbf{v} = \mathbf{A}\mathbf{u}$ where $\mathbf{A}$ is a $N \times N$ matrix, and $\mathbf{u}$, $\mathbf{v}$ are vectors of length N, can be partitioned in such a way that the first N/p rows of matrix $\mathbf{A}$ are only known to process 0, as are the first N/p elements of the vectors $\mathbf{u}$ and $\mathbf{v}$. If we are performing a simple matrix-vector calculation using row-wise partitioning over 3 processes then it can be see from the schematic

$$
\begin{array}{c}
\text{Proc}_0\left\{\begin{array}{c}\\ \\ \\ \end{array}\right.\\
\text{Proc}_1\left\{\begin{array}{c}\\ \\ \end{array}\right.\\
\text{Proc}_2\left\{\begin{array}{c}\\ \\ \end{array}\right.
\end{array}
\begin{pmatrix} v_0\\ v_1\\ \vdots \\ \hline \vdots \\ \hline \vdots \\ v_{N-1}\end{pmatrix}
=
\begin{pmatrix} A_{00} & A_{01} & A_{02} & \cdots & A_{0,N-1}\\ A_{10} & A_{11} & A_{12} & \cdots & A_{1,N-1}\\ \vdots & \ddots & \ddots & \ddots & \vdots\\ \hline \vdots & \ddots & \ddots & \ddots & \vdots\\ \hline \vdots & \ddots & \ddots & \ddots & \vdots\\ A_{N-1,0} & A_{N-1,1} & A_{N-1,2} & \cdots & A_{N-1,N-1}\end{pmatrix}
\begin{pmatrix} u_0\\ u_1\\ \vdots\\ \hline \vdots\\ \hline \vdots\\ u_{N-1}\end{pmatrix},
$$

that in order for process 0 to compute the first N/p elements of $\mathbf{v}$ it is required to know only the first N/p rows of $\mathbf{A}$ (which are held locally) and *all* the elements of $\mathbf{u}$ (most of which are not local to process 0).

More generally, in order to solve the linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ using an iterative approach (such as the conjugate gradient method described in Sect. A.2.3) there are a limited number of operations which will be needed:

- scalar-vector multiplication—an operation on locally-held data;
- vector-vector addition and subtraction—operations on locally-held data;
- a vector Euclidean norm—a sum of squares on local data, followed by a global sum of squares (a parallel reduction), followed by a square-root; and
- matrix-vector multiplication—in which, as outlined above, data from the vector must be communicated between all the processes.

We illustrate an implementation of this fashion of parallel linear algebra by giving a bare-bones working `MpiVector` class. This class contains the features listed below, which will aid building a parallel conjugate gradient solver.

- On constructing a vector of size N, the components are automatically distributed between p processes. Each process is assigned N/p elements. This division may be rounded down so that there will be a shortfall in cases where p does not divide

N. This shortfall is picked up by the top–most process. Each process holds mSize elements corresponding to indices in the range mLo $\leq i <$ mHi.
- There is an overloaded [] operator for accessing elements of the vector. This operator converts between a *global index* into the vector and the *local index* into the process' private data. Any out-of-range indexing trips an assertion.
- Helper methods GetHi and GetLo enable the caller to probe the range of locally held data, thus ameliorating the fact that the partitioning code is hidden from the caller which would make it easy to trip index violation assertions.
- There is a CalculateNorm method which calculates the 2-norm (see Sect. A.1.5) by calculating a local sum of squares, using reduction to sum the local sums into a global sum, and taking the square root. Note the use of Allreduce which ensures that the result of the reduction (and therefore of the norm) is available to all processes.
- There is a method UpdateGlobal for *gathering* all elements of the vector from the remote processes.

The method UpdateGlobal uses more than one gather operation as introduced in Sect. 11.4.2.4, and gathers the entire vector into private storage on every process. The first two gather operations assemble information about the number of locally held data and their displacements. These operations are here to illustrate a common use of fixed- and variable-sized gathers but they are redundant for multiple reasons: (i) the sizes and displacements are fixed in constructor and do not need to be re-calculated on every communication, (ii) the sizes and displacements are not independent—one can be calculated from the other, (iii) the algorithm for calculating sizes and displacements in the constructor is quite simple and could be repeated here.

```
#include <mpi.h>
#include <cmath>
#include <cassert>

class MpiVector
{
private:
   //Store components
   int mLo, mHi, mSize;
   double* mData;
   double* mGlobalData;
public:
   MpiVector(int vecSize)
   {
      int num_procs = MPI::COMM_WORLD.Get_size();
      int rank =  MPI::COMM_WORLD.Get_rank();
      int ideal_local_size = vecSize/num_procs;

      assert (ideal_local_size > 0);
      mLo = ideal_local_size * rank;
      mHi = ideal_local_size * (rank+1);

      //Top processor picks up extras
      if (rank == num_procs-1)
      {
         mHi = vecSize;
      }
      assert(mHi > mLo);
      mData = new double[mHi - mLo];
      mGlobalData =  new double[vecSize];
      mSize = vecSize;
   }
   ~MpiVector()
   {
      delete[] mData;
      delete[] mGlobalData;
   }

   double& operator[](int globalIndex)
   {
      //Make sure that this on the local vector
      assert(mLo<=globalIndex && globalIndex<mHi);
      return mData[globalIndex-mLo];
   }

   int GetHi()
   {
      return mHi;
   }

   int GetLo()
   {
      return mLo;
   }

   double CalculateNorm() const
   {
      double local_sum = 0.0;
      for (int i=0; i<mHi-mLo; i++)
      {
         local_sum += mData[i]*mData[i];
      }
      double global_sum;
      MPI::COMM_WORLD.Allreduce(&local_sum, &global_sum, 1,
                                MPI::DOUBLE, MPI::SUM);
      return sqrt(global_sum);
   }
   void UpdateGlobal()
   {
      int num_procs = MPI::COMM_WORLD.Get_size();

      int* num_per_proc = new int[num_procs];
      int local_size = mHi-mLo;
      MPI::COMM_WORLD.Allgather(&local_size, 1, MPI::INT,
                                num_per_proc, 1, MPI::INT);

      int* lows_per_proc = new int[num_procs];
      MPI::COMM_WORLD.Allgather(&mLo, 1, MPI::INT,
                                lows_per_proc, 1, MPI::INT);

      MPI::COMM_WORLD.Allgatherv(mData, local_size,
                     MPI::DOUBLE, mGlobalData, num_per_proc,
                     lows_per_proc, MPI::DOUBLE);
      delete [] num_per_proc;
      delete [] lows_per_proc;
   }
};
```

```
#include <iostream>
#include <mpi.h>
#include "MpiVector.hpp"

int main(int argc, char* argv[])
{
   MPI::Init(argc, argv);
   MpiVector all_ones(9);
   std::cout << "Local has [" << all_ones.GetLo() <<
                ", " << all_ones.GetHi() << ")\n";
   for (int i=all_ones.GetLo(); i<all_ones.GetHi(); i++)
   {
      all_ones[i] = 1.0;
   }
   assert( fabs(all_ones.CalculateNorm()-3.0) < 1.0e-6 );

   all_ones.UpdateGlobal();
   MPI::Finalize();
   return 0;
}
```

11.6 Tips: Debugging a Parallel Program

We have discussed debugging sequential code in Sects. 1.7 and 7.7. Message passing clearly introduces the potential for different errors to be inserted into your code. We discuss some methods for debugging parallel programs below.

11.6.1 Tip 1: Make an Abstract Program

As with sequential programming, it is very rare for a programmer to begin building a parallel program from scratch. In many cases, you may be given a sequential program which has been written by someone else, or you may be starting from your own program. At such times, it is hard to see the communication patterns underlying the parallel code—they can easily get lost in the details of the calculations.

Our advice is to first take the time to design a rough idea of the communication patterns needed in your new parallel program, and then start afresh. Write a simplified *abstract program* which concentrates on the communication, but neglects the main calculation. This will give you the opportunity to ensure the safe working of the parallel communication in the absence of details of the particulars. Once the message passing is working correctly, it can easily be integrated into the main code.

11.6.2 Tip 2: Datatype Mismatch

In the following code, copied incorrectly from Listing 11.2, the process 0 block has been amended so that the type of the data is now `int` and the message is sent as `MPI::INT`. However, this change has not been reflected in the code for the receiving process where the message is received as `MPI::DOUBLE`.

```
int tag = 30;
if (MPI::COMM_WORLD.Get_rank() == 0)
{
   //Specific send code for process 0
   int send_buffer[2] = {100, 200};
   MPI::COMM_WORLD.Send(send_buffer, 2,
                        MPI::INT, 1, tag);
}
if (MPI::COMM_WORLD.Get_rank() == 1)
{
   //Specific receive code for process 1
   double recv_buffer[2] = {0.0, 0.0};
   MPI::COMM_WORLD.Recv(recv_buffer, 2, MPI::DOUBLE,
                        MPI::ANY_SOURCE, MPI::ANY_TAG);
   std::cout << recv_buffer[0] << "\n";
   std::cout << recv_buffer[1] << "\n";
}
```

The message passing in this program may work correctly—in terms of the communication pattern—but the data received on process 1 will probably be incorrect. This may be because of mismatches in the *size* of the data (on most architectures `int` uses 32 bits whereas `double` uses 64 bits) or it may be due to errors in the *conversion* of the data.

Problems where message data types (or sizes) do not match can be hard to see, especially when the send and receive components are in separate methods or in separate files.

11.6.3 Tip 3: Intermittent Deadlock

Deadlock is the technical term for the situation in which all processes are waiting for some event to happen before proceeding but no process can supply that event because they are waiting for another process. This situation is illustrated simply by four cars arriving simultaneously at a junction where the traffic signals have failed: with nothing to tell them how to proceed all four drivers play safe and wait for someone else to make the first move. In most cases, it is possible to find code which causes deadlock by heavily instrumenting the program, that is, by printing out lots of information and flushing the output. We will deliberately induce deadlock in Exercise 11.2 by never receiving any sent messages so that eventually the sender is not able to proceed because it is unable to send any more messages.

Problems involving *intermittent deadlock* are harder to diagnose. These are situations where the program deadlocks on some runs of the code but runs normally on others. Perhaps the program runs without encountering problems with some trivial example test input, but when it is fed with the real-life input it then deadlocks. When this happens, it is an indication that the problem is to do with the size or timing of messages. In Exercise 11.2 we demonstrate that small amounts of data can be buffered—which hides the fact that a non-buffered blocking send would produce deadlock—but large amounts of data cannot be buffered. In other words, for a given program there may be sizes and timings of messages where deadlock happens, and some where it does not happen.

In such situations, a good strategy is to concentrate on those situations most likely to deadlock. We make our program less efficient and more likely to deadlock by removing buffering and asynchronous messages: replacing all instances of `Send` with `Ssend`. Once all message passing is synchronous it is likely that the intermittent deadlock has become predictable deadlock, allowing us to identify the problem and debug the code. A program can also be made "more synchronous" by splitting calculation steps up with barriers. The program can later be made more efficient as necessary.

11.6.4 Tip 4: Almost Collective Communication

It is common to treat process zero as a "master process", orchestrating the tasks of the other processes, reducing data for output to the screen, and gathering information from all processes for output to a single file. In these circumstances, it is usual to have some blocks of code or some methods which are only executed by the master process and some which are only executed by the "slave processes".

In Sect. 11.4.2.4, we gave the example of an output pattern in which all data was *concentrated* onto a single process before being written to disk. In this case, process zero may execute a block of code consisting of receives and writes to disk via an `ofstream`, whereas the other processes execute a block consisting of the matching send commands. When debugging parallel code, it is usually a good idea to add barriers in order to break the program into manageable sections. However, if we were to add barriers into the slave processes' block of sending code, this would be a recipe for instant deadlock. Since all processes *except one* are executing this code, then any collective communication on `MPI::COMM_WORLD` cannot complete. If collective communication is necessary in this code, then a new communication group (including all processes in `MPI::COMM_WORLD` except rank zero) must be created. New communication groups can be created using relevant MPI functions such as `MPI_Comm_split` (see MPI documentation for more details).

11.7 Exercises

11.1 Amend the `MpiHelloWorld` program in Listing 11.1 so that the processes print in reverse rank order. You can do this with a down-loop over processes and a barrier. Beware that if your implementation of MPI buffers output then you might not be able to verify that your process is working correctly!

Assuming that your loop for output is correct, now modify it to do *round robin* file output. Instead of writing process ranks to `std::cout` each process in turn should: open a named file, write the rank information to it and close the file. The second process to write (and those subsequent) should not open the file until the previous process has closed it and should open the file in append mode (see Sect. 3.2).

Investigate the `MPI::Wtime` method (which returns a high-precision time, with units of seconds, since some fixed point of time in the past) and use it to time the program on each process. Use `Reduce` to compute the average duration of the program over all processes.

11.2 The MPI standard allows the `Send` library call to behave either like a buffered send or like a blocking send. In practice, all implementations of the MPI standard treat `Send` the same way. If the message is small enough (and there is space), then it is copied into a private buffer, and the MPI library is delegated to ensure that the message is delivered and the program flow continues—similar to `Bsend`. If the message is large (or if that private buffer is full), then delivery of the message must wait until the recipient is ready for it, so the program flow waits—similar to `Ssend`.

Write an MPI program where the master process has one loop which attempts to send larger messages each time, and then prints how big the message was. We suggest that you double the size of the message on each iteration. All other processes should do nothing. We suggest that you have an array of length *at least* a million items, to make sure that there is always something to be sent. Eventually you should observe deadlock.

11.3 Write an MPI code following the instructions below. This code is to be executed with only two processes, and tests the use of MPI for transferring vectors of data between processes.

- Define an array `V[10][10]` to store the entries of a 10×10 matrix. The process with rank 0 initialises its copy of the array to

  ```
  V[row][col] = 10*row+col,
  ```

  while the process with rank 1 initialises its copy of the array to

  ```
  V[row][col] = 100+10*row+col.
  ```

  This choice provides a convenient way of identifying, from the value of the entry of `V`, where it has come from in the original arrays, and from which process: the three-digit value `xyz` will be row `y`, column `z` from process `x`.
- Transfer the data stored in the first row of the matrix stored by process 0 into the corresponding positions in the matrix stored by process 1. This involves process 0 sending the data using `Send`, and process 1 receiving the data using `Recv`. One way of doing this on the sending side is to first copy the data into a buffer vector of suitable length and then send this vector. Similarly, on the receiving side receive it into a buffer vector of suitable length and then copy it into the appropriate part of `V`.
- Print out the contents of the array `V` stored by process 1 to check that you have correctly sent the data.
- Repeat the transfer of the first row of data without copying into a buffer on the sender or copying from a buffer on the receiver.
- Repeat the transfer of data sending both the row with index 5 and the row with index 8 between the processes.
- Transfer the first *column* of data between the processes.

11.4 The aim of this exercise is to get you started on writing algorithms with collective communications. The exercise asks you to develop a parallel algorithm for calculating an approximation to π using Monte Carlo integration.

Suppose we want to approximate the integral

$$\int_a^b f(x)\ \mathrm{d}x,$$

where $f(x)$ is a continuous function defined at all points in the closed interval $a \le x \le b$. If X_i, $i = 0, 1, 2, \ldots, N-1$ are independent random variables uniformly distributed on the interval $a \le x \le b$, where N is sufficiently large, then Monte Carlo integration allows us to approximate the integral by

$$\int_a^b f(x)\ \mathrm{d}x \approx \frac{b-a}{N} \sum_{i=0}^{N-1} f(X_i).$$

Noting that

$$\pi = 4\int_0^1 \frac{1}{1+x^2}\mathrm{d}x,$$

we will use Monte Carlo integration to estimate π through approximating the integral on the right-hand side of this equation. Sequential code for this is given below.

The random numbers are generated through the random number generator `rand` (line 18), and seeded through `srand` (line 11). The random number generator requires the `cstdlib` header to be included. The random number generator is seeded differently on every run: in this exercise you will develop this code to run on a distributed memory machine through use of MPI statements, and you don't want a set of parallel computers to all work on the same set of "random" numbers.

```
1  // Compute pi using Monte Carlo integration
2  // of 1/(1+x*x) on the interval 0<=x<=1
3  #include <cmath>
4  #include <cstdlib>
5  #include <iostream>
6  #include <unistd.h> //For getpid()
7
8  int main(int argc, char* argv[])
9  {
10    // seed random number generator
11    srand(getpid());
12    int n_points = 1000000;
13
14    double sum = 0;
15    for (int i=0; i<n_points; i++)
16    {
17       // generate a random number on the interval 0<=x<=1
18       double x = rand()/((double)(RAND_MAX));
19       double f = 1.0/(1.0+x*x);
20       sum += f;
21    }
22    double pi = 4.0*(sum/((double)(n_points)));
23    std::cout << "Pi is approximately " << pi
24              << " with error " << pi-M_PI  << "\n";
25
26    return 0;
27 }
```

Compile the program and it should print out an answer similar to

```
Pi is approximately 3.141782 with error 0.000355562
```

In the exercises below, we will add MPI function calls to enable this code to be run in parallel.

1. Add MPI function calls so that `n_points` function evaluations are performed on each of the MPI processes.
2. Estimate π through reducing the result of function evaluations (`sum`) from each processor to a global sum on process 0 and scaling appropriately. This is similar to the summation of a series in Sect. 11.5.1.
3. Amend the code so that process 0 selects a value of `n_points` for each of the processes at the beginning program. Pass these values out in a scatter operation.

11.5 Write classes to enable parallel linear algebra based on the row-wise matrix partitioning—and the `MpiVector` class—given Sect. 11.5.2. Your goal for this exercise should be to perform a matrix-vector multiplication in parallel.

1. Add as much functionality and overloaded operators from the `Vector` class given in Sect. 10.1 to `MpiVector` as you wish. Include any improvements which you may have made to `Vector` as part of Exercise 10.2.
2. The `MpiVector` constructor contains an assertion that the ideal local size (number of local vector elements) should be nonzero. This guards against the case when the number of processes is larger than `vecSize`, in which case the current code in the constructor would assign the entire vector to the top-most process. Fix this situation so that when there are fewer vector elements than processes every process is assigned either one or zero elements.
3. Make it possible to set elements on remote processes. A suitable scheme would be to construct the vector in “set up” mode, during which requests to add values to remote elements are stored for later. A user is able to call a method `FinishSetUp` which communicates the stored data between processes, puts the vector in a “usable” mode and bars future attempts to set remote data.
4. Remove some of the redundant calculations performed by `UpdateGlobal` mentioned in Sect. 10.1.
5. Write an output method which uses `UpdateGlobal` such that one process is able to print the entire vector to screen or to file.
6. The `UpdateGlobal` method relies on memory for the private data member `mGlobalData` being allocated in the constructor. Since the `mGlobalData` is only required for output or for a matrix-vector product, the memory for `mGlobalData` ought to be allocated on demand. Make sure that there is also a method for de-allocating this memory when it is no longer needed.
7. Write an `MpiMatrix` class using the scheme outlined in Sect. 10.1. It is important that you treat the partition on the number of matrix rows in exactly the same way as the vector partition, so that local sizes are always compatible. Perform a matrix-vector multiplication in parallel and output the solution.