Writing Message Passing Parallel Programs with MPI

A Two Day Course on MPI Usage

Course Notes

Version 1.8.2

Neil MacDonald, Elspeth Minty, Joel Malard, Tim Harding, Simon Brown, Mario Antonioletti

Edinburgh Parallel Computing Centre

The University of Edinburgh

epcc

Table of Contents

1 The MPI Interface

In principle, a sequential algorithm is portable to any architecture supporting the sequential paradigm. However, programmers require more than this: they want their realisation of the algorithm in the form of a particular *program* to be portable — source-code portability.

The same is true for message-passing programs and forms the motivation behind MPI. MPI provides source-code portability of message-passing programs written in C or Fortran across a variety of architectures. Just as for the sequential case, this has many benefits, including

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

While the basic concept of processes communicating by sending messages to one another has been understood for a number of years, it is only relatively recently that message-passing systems have been developed which allow source-code portability.

MPI was the first effort to produce a message-passing interface standard across the whole parallel processing community. Sixty people representing forty different organisations — users and vendors of parallel systems from both the US and Europe — collectively formed the “MPI Forum”. The discussion was open to the whole community and was led by a working group with in-depth experience of the use and design of message-passing systems (including PVM, PARMACS, and EPCC’s own CHIMP). The two-year process of proposals, meetings and review resulted in a document specifying a standard *M*essage *P*assing *I*nterface (MPI).

1.1 Goals and scope of MPI

MPI’s prime goals are:

- To provide source-code portability
- To allow efficient implementation across a range of architectures

It also offers:

- A great deal of functionality
- Support for heterogeneous parallel architectures

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

- Initial loading of processes onto processors
- Spawning of processes during execution
- Debugging
- Parallel I/O

1.2 Preliminaries

MPI comprises a library. An MPI process consists of a C or Fortran 77 program which communicates with other MPI processes by calling MPI routines. The MPI routines provide the programmer with a consistent interface across a wide variety of different platforms.

The initial loading of the executables onto the parallel machine is outwith the scope of the MPI interface. Each implementation will have its own means of doing this. Appendix A :"Compiling and Running MPI Programs on lomond" on page 73 contains information on running MPI programs on `lomond`. More general information on `lomond` can be found in the "Introduction to the University of Edinburgh HPC Service" document.

The result of mixing MPI with other communication methods is undefined, but MPI is guaranteed not to interfere with the operation of standard language operations such as `write`, `printf` etc. MPI may (with care) be mixed with OpenMP, but the programmer may *not* make the assumption that MPI is thread-safe, and *must* make sure that any necessary explicit synchronisation to force thread-safety is carried out by the program.

1.3 MPI Handles

MPI maintains internal data-structures related to communications etc. and these are referenced by the user through *handles*. Handles are returned to the user from some MPI calls and can be used in other MPI calls.

Handles can be copied by the usual assignment operation of C or Fortran.

1.4 MPI Errors

In general, C MPI routines return an `int` and Fortran MPI routines have an `IERROR` argument — these contain the error code. The default action on detection of an error by MPI is to cause the parallel computation to abort, rather than return with an error code, but this can be changed as described in "Error Messages" on page 63.

Because of the difficulties of implementation across a wide variety of architectures, a complete set of detected errors and corresponding error codes is not defined. An MPI program might be *erroneous* in the sense that it does not call MPI routines correctly, but MPI does not guarantee to detect all such errors.

1.5 Bindings to C and Fortran 77

All names of MPI routines and constants in both C and Fortran begin with the prefix `MPI_` to avoid name collisions.

Fortran routine names are all upper case but C routine names are mixed case — following the MPI document [1], when a routine name is used in a language-independent context, the upper case version is used. All constants are in upper case in both Fortran and C.

In Fortran[1], handles are always of type `INTEGER` and arrays are indexed from `1`.

1. Note that although MPI is a Fortran 77 library, at EPCC MPI programs are usually compiled using a Fortran 90 compiler. As Fortran 77 is a sub-set of Fortran 90, this is quite acceptable.

In C, each type of handle is of a different `typedef`'d type (`MPI_Datatype`, `MPI_Comm`, etc.) and arrays are indexed from `0`.

Some arguments to certain MPI routines can legitimately be of any type (`integer`, `real` etc.). In the Fortran examples in this course

```
MPI_ROUTINE (MY_ARGUMENT, IERROR)

<type> MY_ARGUMENT
```

indicates that the type of `MY_ARGUMENT` is immaterial. In C, such arguments are simply declared as `void *`.

1.6 Initialising MPI

The first MPI routine called in any MPI program *must* be the initialisation routine `MPI_INIT`[1]. Every MPI program must call this routine *once*, before any other MPI routines. Making multiple calls to `MPI_INIT` is erroneous. The C version of the routine accepts the arguments to `main`, `argc` and `argv` as arguments.

```
int MPI_Init(int *argc, char ***argv);
```

The Fortran version takes no arguments other than the error code.

```
MPI_INIT(IERROR)

  INTEGER IERROR
```

1.7 `MPI_COMM_WORLD` and communicators

`MPI_INIT` defines something called `MPI_COMM_WORLD` for each process that calls it. `MPI_COMM_WORLD` is a *communicator*. All MPI communication calls require a communicator argument and MPI processes can only communicate if they share a communicator.

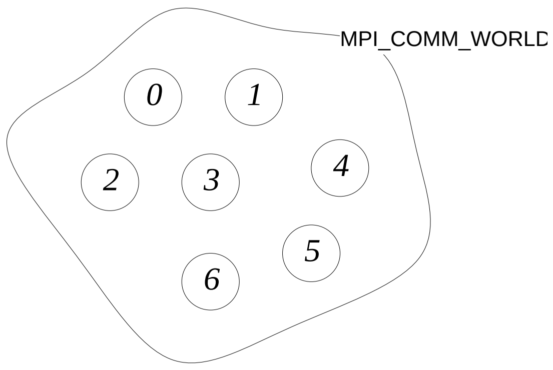

Figure 1: The predefined communicator `MPI_COMM_WORLD` *for seven processes. The numbers indicate the ranks of each process.*

Every communicator contains a *group* which is a list of processes. Secondly, a group is in fact *local* to a particular process. The apparent contradiction between this statement and that in the text is explained thus: the group contained within a communicator has been previously agreed across the processes at the time when the communicator was

1.There is in fact one exception to this, namely `MPI_INITIALIZED` which allows the programmer to test whether `MPI_INIT` has already been called.

set up. The processes are ordered and numbered consecutively from `0` (in both Fortran and C), the number of each process being known as its *rank*. The rank identifies each process within the communicator. For example, the rank can be used to specify the source or destination of a message. (It is worth bearing in mind that in general a process could have several communicators and therefore might belong to several groups, typically with a different rank in each group.) Using `MPI_COMM_WORLD`, every process can communicate with every other. The group of `MPI_COMM_WORLD` is the set of all MPI processes.

1.8 Clean-up of MPI

An MPI program should call the MPI routine `MPI_FINALIZE` when all communications have completed. This routine cleans up all MPI data-structures etc. It does not cancel outstanding communications, so it is the responsibility of the programmer to make sure all communications have completed. Once this routine has been called, no other calls can be made to MPI routines, not even `MPI_INIT`, so a process cannot later re-enrol in MPI.

```
MPI_FINALIZE()
```
[1]

1.9 Aborting MPI

```
MPI_ABORT(comm, errcode)
```

This routine attempts to abort all processes in the group contained in `comm` so that with `comm = MPI_COMM_WORLD` the whole parallel program will terminate.

1.10 A simple MPI program

All MPI programs should include the standard header file which contains required defined constants. For C programs the header file is `mpi.h` and for Fortran programs it is `mpif.h`. Taking into account the previous two sections, it follows that *every* MPI program should have the following outline.

1.10.1 C version

```
#include <mpi.h>

/* Also include usual header files */

main(int argc, char **argv)

  {

    /* Initialise MPI */

    MPI_Init (&argc, &argv);

    /* There is no main program */

    /* Terminate MPI */

    MPI_Finalize ();
```

1.The C and Fortran versions of the MPI calls can be found in the MPI specification provided.

```
     exit (0);

  }
```

1.10.2 Fortran version

```
      PROGRAM simple

      include 'mpif.h'

      integer errcode

C Initialise MPI

      call MPI_INIT (errcode)

C The main part of the program goes here.

C Terminate MPI

      call MPI_FINALIZE (errcode)

      end
```

1.10.3 Accessing communicator information

An MPI process can query a communicator for information about the group, with `MPI_COMM_SIZE` and `MPI_COMM_RANK`.

```
MPI_COMM_RANK (comm, rank)
```

`MPI_COMM_RANK` returns in `rank` the rank of the calling process in the group associated with the communicator `comm`.

`MPI_COMM_SIZE` returns in `size` the number of processes in the group associated with the communicator `comm`.

```
MPI_COMM_SIZE (comm, size)
```

1.11 Exercise: Hello World - the minimal MPI program

1. Write a minimal MPI program which prints the message "Hello World". Compile and run it on a single processor.
2. Run it on several processors in parallel.
3. Modify your program so that only the process ranked 0 in `MPI_COMM_WORLD` prints out the message.
4. Modify your program so that the number of processes (ie: the value of `MPI_COMM_SIZE`) is printed out.

Extra exercise

What happens if you omit the last MPI procedure call in your MPI program?

2 What’s in a Message?

An MPI message is an array of elements of a particular MPI *datatype*.

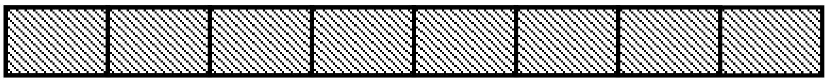

Figure 2: An MPI message.

All MPI messages are *typed* in the sense that the type of the contents must be specified in the send and receive. The basic datatypes in MPI correspond to the basic C and Fortran datatypes as shown in the tables below.

Table 1: Basic C datatypes in MPI

MPI Datatype	C datatype
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	

Table 2: Basic Fortran datatypes in MPI

MPI Datatype	Fortran Datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

There are rules for datatype-matching and, with certain exceptions, the datatype specified in the receive must match the datatype specified in the send. The great advantage of this is that MPI can support *heterogeneous* parallel architectures i.e. parallel machines built from different processors, because type conversion can be performed when necessary. Thus two processors may represent, say, an integer in different ways, but MPI processes on these processors can use MPI to send integer messages without being aware of the heterogeneity[1]

More complex datatypes can be constructed at run-time. These are called *derived* datatypes and are built from the basic datatypes. They can be used for sending strided vectors, C `structs` etc. The construction of new datatypes is described later. The MPI datatypes `MPI_BYTE` and `MPI_PACKED` do not correspond to any C or Fortran datatypes. `MPI_BYTE` is used to represent eight binary digits and `MPI_PACKED` has a special use discussed later.

1.Whilst a single implementation of MPI may be designed to run on a parallel "machine" made up of heterogeneous processors, there is no guarantee that two different MPI implementation can successfully communicate with one another — MPI defines an interface to the programmer, but does not define message protocols etc.

3 Point-to-Point Communication

3.1 Introduction

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

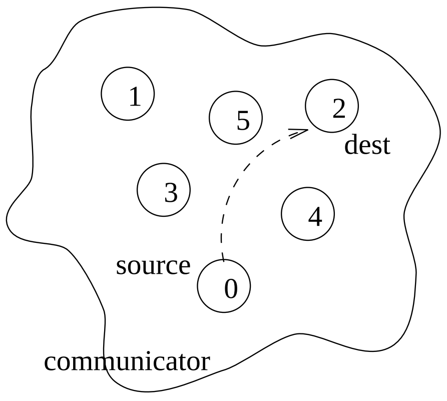

Figure 3: In point-to-point communication a process sends a message to another specific process

To send a message, a *source* process makes an MPI call which specifies a *destination* process in terms of its rank in the appropriate communicator (e.g. `MPI_COMM_WORLD`). The destination process also has to make an MPI call if it is to receive the message.

3.2 Communication Modes

There are four *communication modes* provided by MPI: *standard*, *synchronous*, *buffered* and *ready*. The modes refer to four different types of *send*. It is not meaningful to talk of communication mode in the context of a receive. "Completion" of a send means by definition that the send buffer can safely be re-used. The standard, synchronous and buffered sends differ only in one respect: how completion of the send depends on the *receipt* of the message.

Table 3: MPI communication modes

	Completion condition
Synchronous send	Only completes when the receive has completed.

Table 3: MPI communication modes

	Completion condition
Buffered send	Always completes (unless an error occurs), irrespective of whether the receive has completed.
Standard send	Either synchronous or buffered.
Ready send	Always completes (unless an error occurs), irrespective of whether the receive has completed.
Receive	Completes when a message has arrived.

All four modes exist in both blocking and non-blocking forms. In the blocking forms, return from the routine implies completion. In the non-blocking forms, all modes are tested for completion with the usual routines (`MPI_TEST`, `MPI_WAIT`, etc.)

Table 4: MPI Communication routines

	Blocking form
Standard send	MPI_SEND
Synchronous send	MPI_SSEND
Buffered send	MPI_BSEND
Ready send	MPI_RSEND
Receive	MPI_RECV

There are also “persistent” forms of each of the above, see “Persistent communications” on page 66.

3.2.1 Standard Send

The standard send completes once the message has been sent, which *may or may not* imply that the message has arrived at its destination. The message may instead lie “in the communications network” for some time. A program using standard sends should therefore obey various rules:

- It should not assume that the send will complete *before* the receive begins. For example, two processes should not use blocking standard sends to exchange messages, since this may on occasion cause deadlock.
- It should not assume that the send will complete *after* the receive begins. For example, the sender should not send further messages whose correct interpretation depends on the assumption that a previous message arrived elsewhere; it is possible to imagine scenarios (necessarily with more than two processes) where the ordering of messages is non-deterministic under standard mode.

  In summary, a standard send may be implemented as a synchronous send, or it may be implemented as a buffered send, and the user should not assume either case.
- Processes should be *eager readers*, i.e. guarantee to eventually receive all messages sent to them, else the network may overload.

If a program breaks these rules, unpredictable behaviour can result: programs may run successfully on one implementation of MPI but not on others, or may run successfully on some occasions and "hang" on other occasions in a non-deterministic way.

The standard send has the following form

```
MPI_SEND (buf, count, datatype, dest, tag, comm)
```

where

- `buf` is the address of the data to be sent.
- `count` is the number of elements of the MPI datatype which `buf` contains.
- `datatype` is the MPI datatype.
- `dest` is the destination process for the message. This is specified by the rank of the destination process within the group associated with the communicator `comm`.
- `tag` is a marker used by the sender to distinguish between different types of messages. Tags are used by the programmer to distinguish between different sorts of message.
- `comm` is the communicator shared by the sending and receiving processes. Only processes which have the same communicator can communicate.
- `IERROR` contains the return value of the Fortran version of the synchronous send.

Completion of a send means by definition that the send buffer can safely be re-used i.e. the data has been sent.

3.2.2 Synchronous Send

If the sending process needs to know that the message has been received by the receiving process, then both processes may use *synchronous* communication. What actually happens during a synchronous communication is something like this: the receiving process sends back an acknowledgement (a procedure known as a 'handshake' between the processes) as shown in Figure 4:. This acknowledgement must be received by the sender before the send is considered complete.

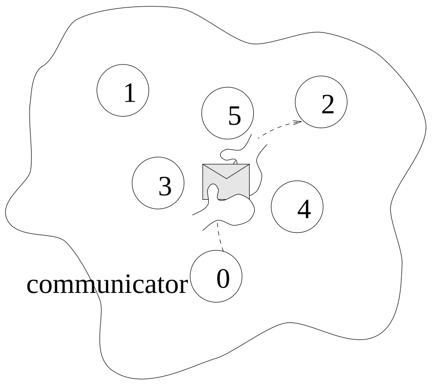

Figure 4: In the synchronous mode the sender knows that the other one has received the message.

The MPI synchronous send routine is similar in form to the standard send. For example, in the blocking form:

```
MPI_SSEND (buf, count, datatype, dest, tag, comm)
```

If a process executing a blocking synchronous send is "ahead" of the process executing the matching receive, then it will be idle until the receiving process catches up. Similarly, if the sending process is executing a non-blocking synchronous send, the completion test will not succeed until the receiving process catches up. Synchronous mode can therefore be slower than standard mode. Synchronous mode is however a *safer* method of communication because the communication network can never become overloaded with undeliverable messages. It has the advantage over standard mode of being more predictable: a synchronous send always synchronises the sender and receiver, whereas a standard send may or may not do so. This makes the behaviour of a program more deterministic. Debugging is also easier because messages cannot lie undelivered and "invisible" in the network. Therefore a parallel program using synchronous sends need only take heed of the rule on page 10. Problems of unwanted synchronisation (such as deadlock) can be avoided by the use of non-blocking synchronous communication "Non-Blocking Communication" on page 19.

3.2.3 Buffered Send

Buffered send guarantees to complete immediately, copying the message to a system buffer for later transmission if necessary. The advantage over standard send is predictability — the sender and receiver are guaranteed *not* to be synchronised and if the network overloads, the behaviour is defined, namely an error will occur. Therefore a parallel program using buffered sends need only take heed of the rule on page 10. The disadvantage of buffered send is that the programmer cannot assume any pre-allocated buffer space and must explicitly attach enough buffer space for the program with calls to `MPI_BUFFER_ATTACH`. Non-blocking buffered send has no advantage over blocking buffered send.

To use buffered mode, the user must attach buffer space:

```
MPI_BUFFER_ATTACH (buffer, size)
```

This specifies the array `buffer` of `size` bytes to be used as buffer space by buffered mode. Of course `buffer` must point to an existing array which will not be used by the programmer. Only one buffer can be attached per process at a time. Buffer space is detached with:

```
MPI_BUFFER_DETACH (buffer, size)
```

Any communications already using the buffer are allowed to complete before the buffer is detached by MPI.

C users note: this does not deallocate the memory in `buffer`.

Often buffered sends and non-blocking communication are alternatives and each has pros and cons:

- buffered sends require extra buffer space to be allocated and attached by the user;
- buffered sends require copying of data into and out of system buffers while non-blocking communication does not;
- non-blocking communication requires more MPI calls to perform the same number of communications.

3.2.4 Ready Send

A ready send, like buffered send, completes immediately. The communication is guaranteed to succeed normally if a matching receive is already posted. However, unlike all other sends, if no matching receive has been posted, the outcome is undefined. As shown in Figure 5:, the sending process simply throws the message out onto the communication network and hopes that the receiving process is waiting to catch it. If the receiving process is ready for the message, it will be received, else the message may be silently dropped, an error may occur, etc.

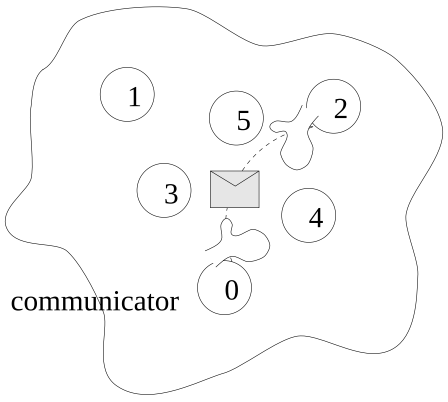

Figure 5: In the ready mode a process hopes that the other process has caught the message

The idea is that by avoiding the necessity for handshaking and buffering between the sender and the receiver, performance may be improved. Use of ready mode is only safe if the logical control flow of the parallel program permits it. For example, see Figure 6:

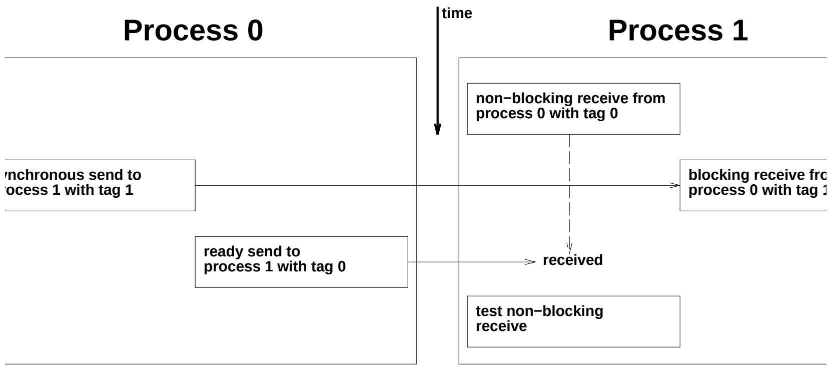
time
Process 0
Process 1
non−blocking receive from process 0 with tag 0
nchronous send to ocess 1 with tag 1
blocking receive fro process 0 with tag 1
ready send to process 1 with tag 0
received
test non−blocking receive

Figure 6: An example of safe use of ready mode. When Process 0 sends the message with tag 0 it ``knows'' that the receive has already been posted because of the synchronisation inherent in sending the message with tag 1.

Clearly ready mode is a difficult mode to debug and requires careful attention to parallel program messaging patterns. It is only likely to be used in programs for which performance is critical and which are targeted mainly at platforms for which there is a real performance gain. The ready send has a similar form to the standard send:

```
MPI_RSEND (buf, count, datatype, dest, tag, comm)
```

Non-blocking ready send has no advantage over blocking ready send (see “Non-Blocking Communication” on page 19).

3.2.5 The standard blocking receive

The format of the standard blocking *receive* is:

```
MPI_RECV (buf, count, datatype, source, tag, comm, status)
```

where

- `buf` is the address where the data should be placed once received (the receive buffer). For the communication to succeed, the receive buffer *must* be large enough to hold the message without truncation — if it is not, behaviour is undefined. The buffer may however be longer than the data received.
- `count` is the number of elements of a certain MPI datatype which `buf` can contain. The number of data elements actually received may be less than this.
- `datatype` is the MPI datatype for the message. This must match the MPI datatype specified in the send routine.
- `source` is the rank of the source of the message in the group associated with the communicator `comm`. Instead of prescribing the source, messages can be received from one of a number of sources by specifying a *wildcard*, `MPI_ANY_SOURCE`, for this argument.
- `tag` is used by the receiving process to prescribe that it should receive only a message with a certain tag. Instead of prescribing the tag, the wildcard `MPI_ANY_TAG` can be specified for this argument.
- `comm` is the communicator specified by both the sending and receiving process. *There is no wildcard option for this argument*.
- If the receiving process has specified wildcards for both or either of `source` or `tag`, then the corresponding information from the message that was actually received may be required. This information is returned in `status`, and can be queried using routines described later.
- `IERROR` contains the return value of the Fortran version of the standard receive.

Completion of a receive means by definition that a message arrived i.e. the data has been received.

3.3 Discussion

The word "blocking" means that the routines described above *only return once the communication has completed*. This is a non-local condition i.e. it might depend on the state of other processes. The ability to select a message by source is a powerful feature. For example, a source process might wish to receive messages back from worker processes in strict order. Tags are another powerful feature. A tag is an integer labelling different types of message, such as "initial data", "client-server request", "results from worker". Note the difference between this and the programmer sending an integer label of his or her own as part of the message — in the latter case, by the time the label is known, the message itself has already been read. The point of tags is that the receiver can select which messages it wants to receive, on the basis of the tag. Point-to-point communications in MPI are led by the sending process "pushing" messages out to other processes — a process cannot "fetch" a message, it can only receive a message if it has been sent. When a point-to-point communication call is made, it is termed *posting* a send or *posting* a receive, in analogy perhaps to a bulletin board. Because of the selection allowed in receive calls, it makes sense to talk of a send matching a receive. MPI can be thought of as an agency — processes post sends and receives to MPI and MPI matches them up.

3.4 Information about each message: the Communication Envelope

As well as the data specified by the user, the communication also includes other information, known as the *communication envelope*, which can be used to distinguish between messages. This information is returned from `MPI_RECV` as `status`.

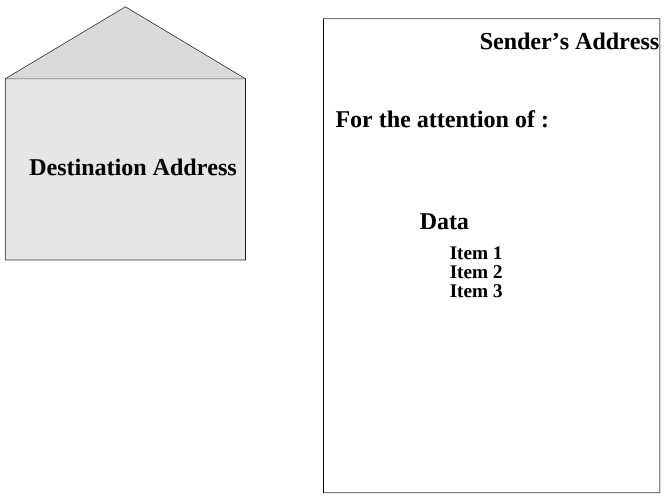

Figure 7: As well as the data, the message contains information about the communication in the communication envelope.

The `status` argument can be queried directly to find out the source or tag of a message which has just been received. This will of course only be necessary if a wildcard option was used in one of these arguments in the receive call. The *source* process of a message received with the `MPI_ANY_SOURCE` argument can be found for C in:

```
status.MPI_SOURCE
```

and for Fortran in:

```
STATUS(MPI_SOURCE)
```

This returns the rank of the source process in the `source` argument. Similarly, the *message tag* of a message received with `MPI_ANY_TAG` can be found for C in:

```
status.MPI_TAG
```

and for Fortran in:

```
STATUS(MPI_TAG)
```

The size of the message received by a process can also be found.

3.4.1 Information on received message size

The message received need not fill the receive buffer. The `count` argument specified to the receive routine is the number of elements for which there is space in the receive buffer. This will not always be the same as the number of elements actually received.

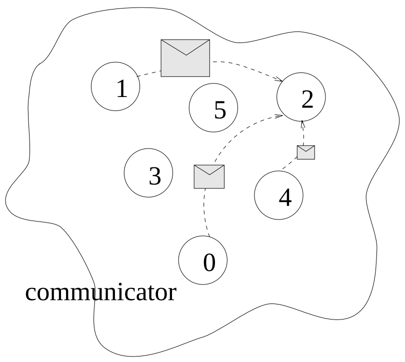

Figure 8: Processes can receive messages of different sizes.

The number of elements which was actually received can be found by querying the communication envelope, namely the `status` variable, after a communication call. For example:

```
MPI_GET_COUNT (status, datatype, count)
```

This routine queries the information contained in `status` to find out how many of the MPI datatype are contained in the message, returning the result in `count`.

3.5 Rules of point-to-point communication

MPI implementations guarantee that the following properties hold for point-to-point communication (these rules are sometimes known as “semantics”).

3.5.1 Message Order Preservation

Messages do not overtake each other. That is, consider any two MPI processes. Process A sends two messages to Process B with the same communicator. Process B posts two receive calls which match both sends. Then the two messages are guaranteed to be received in the order they were sent.

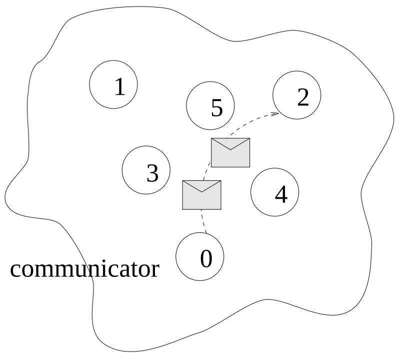

Figure 9: Messages sent from the same sender which match the same receive are received in the order they were sent.

3.5.2 Progress

It is not possible for a matching send and receive pair to remain permanently outstanding. That is, if one MPI process posts a send and a second process posts a matching receive, then either the send or the receive will eventually complete.

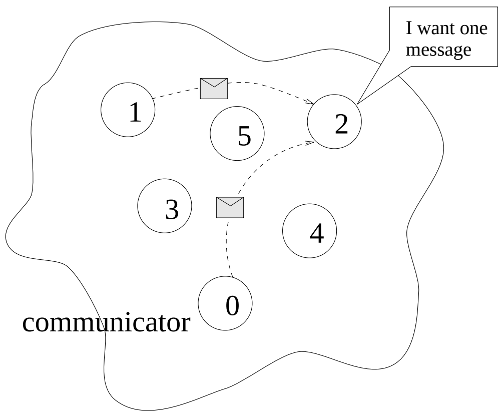

Figure 10: One communication will complete.

There are two possible scenarios:

- The send is received by a third process with a matching receive, in which case the send completes but the second processes receive does not.
- A third process sends out a message which is received by the second process, in which case the receive completes but the first processes send does not.

3.6 Datatype-matching rules

When a message is sent, the receiving process must in general be expecting to receive the same datatype. For example, if a process sends a message with datatype `MPI_INTEGER` the receiving process must specify to receive datatype `MPI_INTEGER`, otherwise the communication is incorrect and behaviour is undefined. Note that this restriction disallows inter-language communication. (There is one exception to this rule: `MPI_PACKED` can match any other type.) Similarly, the C or Fortran type of the variable(s) in the message must match the MPI datatype, *e.g.*, if a process sends a message with datatype `MPI_INTEGER` the variable(s) specified by the process must be of type `INTEGER`, otherwise behaviour is undefined. (The exceptions to this rule are `MPI_BYTE` and `MPI_PACKED`, which, on a byte-addressable machine, can be used to match any variable type.)

3.7 Exercise: Ping pong

1. Write a program in which two processes repeatedly pass a message back and forth.
2. Insert timing calls (see below) to measure the time taken for one message.
3. Investigate how the time taken varies with the size of the message.

3.7.1 Timers

For want of a better place, a useful routine is described here which can be used to time programs.

```
    MPI_WTIME()
```

This routine returns elapsed wall-clock time in seconds. The timer has no defined starting-point, so in order to time something, two calls are needed and the difference should be taken between them.

`MPI_WTIME` is a double-precision routine, so remember to declare it as such in your programs (applies to both C and Fortran programmers). This also applies to variables which use the results returned by `MPI_WTIME`.

Extra exercise

Write a program in which the process with rank 0 sends the same message to all other processes in MPI_COMM_WORLD and then receives a message of the same length from all other processes. How does the time taken varies with the size of the messages and with the number of processes?

4 Non-Blocking Communication

4.1 Example: one-dimensional smoothing

Consider the example in Figure 11: (a simple one-dimensional case of the smoothing operations used in image-processing). Each element of the array must be set equal to the average of its two neighbours, and this is to take place over a certain number of iterations. Each process is responsible for updating part of the array (a common parallel technique for grid-based problems known as *regular domain decomposition*[1]. The two cells at the ends of each process' sub-array are *boundary* cells. For their update, they require boundary values to be communicated from a process owning the neighbouring sub-arrays and two extra *halo* cells are set up to hold these values. The non-boundary cells do not require halo data for update.

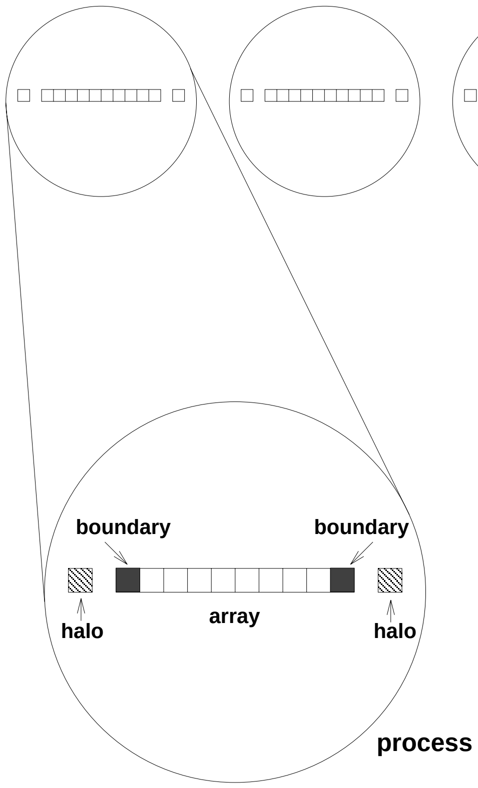

Figure 11: One-dimensional smoothing

1. We use regular domain decomposition as an illustrative example of a particular communication pattern. However, in practice, parallel libraries exist which can hide the communication from the user.

4.2 Motivation for non-blocking communication

The communications described so far are all *blocking* communications. This means that they do not return until the communication has completed (in the sense that the buffer can be used or re-used). Using blocking communications, a first attempt at a parallel algorithm for the one-dimensional smoothing might look like this:

```
for(iterations)

   update all cells;

   send boundary values to neighbours;

   receive halo values from neighbours;
```

This produces a situation akin to that shown in where each process sends a message to another process and then posts a receive. Assume the messages have been sent using a standard send. Depending on implementation details a standard send may not be able to complete until the receive has started. Since *every* process is sending and none is yet receiving, *deadlock* can occur and none of the communications ever complete.

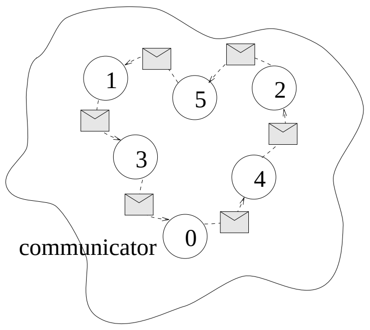

Figure 12: Deadlock

There is a solution to the deadlock based on "red-black" communication in which "odd" processes choose to send whilst "even" processes receive, followed by a reversal of roles[1] — but deadlock is not the only problem with this algorithm. Communication is not a major user of CPU cycles, but is usually relatively slow because of the communication network and the dependency on the process at the other end of the communication. With blocking communication, the process is waiting idly while each communication is taking place. Furthermore, the problem is exacerbated because the communications in each direction are required to take place one after the other. The point to notice is that the non-boundary cells could theoretically be updated during the time when the boundary / halo values are in transit. This is known as latency hiding because the latency of the communications is overlapped with useful work. This requires a decoupling of the completion of each send from the receipt by the neighbour. Non-blocking communication is one method of achieving this.[2] In non-blocking communication the processes call an MPI routine to set up a communi-

1.Another solution might use `MPI_SEND_RECV`

2.It is not the only solution - buffered sends achieve a similar effect.

cation (send or receive), but the routine returns before the communication has completed. The communication can then continue in the background and the process can carry on with other work, returning at a later point in the program to check that the communication has completed successfully. The communication is therefore divided into two operations: the initiation and the completion test. Non-blocking communication is analogous to a form of delegation — the user makes a request to MPI for communication and checks that its request completed satisfactorily only when it needs to know in order to proceed. The solution now looks like:

```
for(iterations)

   update boundary cells;

   initiate sending of boundary values to neighbours;

   initiate receipt of halo values from neighbours;

   update non-boundary cells;

   wait for completion of sending of boundary values;

   wait for completion of receipt of halo values;
```

Note also that deadlock cannot occur and that communication in each direction can occur simultaneously. Completion tests are made when the halo data is required for the next iteration (in the case of a receive) or the boundary values are about to be updated again (in the case of a send)[1].

4.3 Initiating non-blocking communication in MPI

The non-blocking routines have identical arguments to their blocking counterparts except for an extra argument in the non-blocking routines. This argument, `request`, is very important as it provides a handle which is used to test when the communication has completed.

Table 5: Communication models for non-blocking communications

Non-Blocking Operation	MPI call
Standard send	MPI_ISEND
Synchronous send	MPI_ISSEND
Buffered send	MPI_BSEND
Ready send	MPI_RSEND
Receive	MPI_IRECV

1. “Persistent communications” on page 66 describes an alternative way of expressing the same algorithm using persistent communications.

4.3.1 Non-blocking sends

The principle behind non-blocking *sends* is shown in Figure 13:.

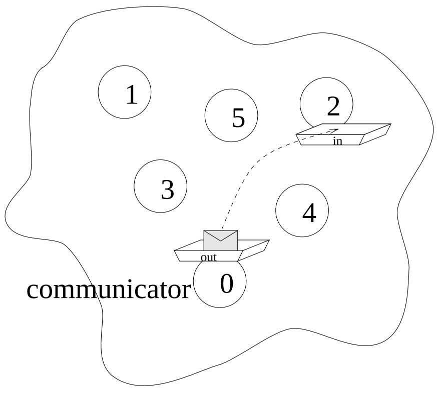

Figure 13: A non-blocking send

The sending process initiates the send using the following routine (in synchronous mode):

```
MPI_ISSEND (buf, count, datatype, dest, tag, comm, request)
```

It then continues with other computations which *do not* alter the send buffer. Before the sending process can update the send buffer it must check that the send has completed using the routines described in “Testing communications for completion” on page 23.

4.3.2 Non-blocking receives

Non-blocking receives may match *blocking* sends and *vice versa.*

A non-blocking receive is shown in Figure 14:.

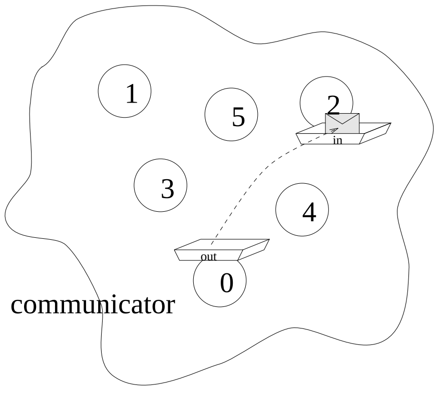

Figure 14: A non-blocking receive

The receiving process posts the following receive routine to initiate the receive:

```
MPI_IRECV (buf, count, datatype, source, tag, comm, request)
```

The receiving process can then carry on with other computations until it needs the received data. It then checks the receive buffer to see if the communication has completed. The different methods of checking the receive buffer are covered in “Testing communications for completion” on page 23.

4.4 Testing communications for completion

When using non-blocking communication it is essential to ensure that the communication has completed before making use of the result of the communication or re-using the communication buffer. Completion tests come in two types:

- `WAIT` type These routines block until the communication has completed. They are useful when the data from the communication is required for the computations or the communication buffer is about to be re-used.

  Therefore a non-blocking communication immediately followed by a `WAIT`-type test is equivalent to the corresponding blocking communication.

- `TEST` type These routines return a TRUE or FALSE value depending on whether or not the communication has completed. They do not block and are useful in situations where we want to know if the communication has completed but do not yet *need* the result or to re-use the communication buffer i.e. the process can usefully perform some other task in the meantime.

4.4.1 Testing a non-blocking communication for completion

The `WAIT`-type test is:

```
MPI_WAIT (request, status)
```

This routine blocks until the communication specified by the handle `request` has completed. The `request` handle will have been returned by an earlier call to a non-blocking communication routine. The `TEST`-type test is:

```
MPI_TEST (request, flag, status)
```

In this case the communication specified by the handle `request` is simply queried to see if the communication has completed and the result of the query (TRUE or FALSE) is returned immediately in `flag`.

4.4.2 Multiple Communications

It is not unusual for several non-blocking communications to be posted at the same time, so MPI also provides routines which test multiple communications at once (see Figure 15:). Three types of routines are provided: those which test for the completion of *all* of the communications, those which test for the completion of *any* of them and those which test for the completion of *some* of them. Each type comes in two forms: the `WAIT` form and the `TEST` form.

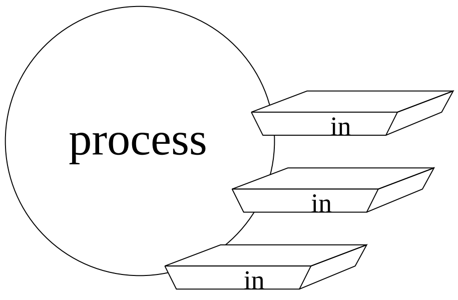

Figure 15: MPI allows a number of specified non-blocking communications to be tested in one go.

The routines may be tabulated:

Table 6: MPI completion routines

Test for completion	WAIT type (blocking)	TEST type (query only)
At least one, return exactly one	MPI_WAITANY	MPI_TESTANY
Every one	MPI_WAITALL	MPI_TESTALL
At least one, return all which completed	MPI_WAITSOME	MPI_TESTSOME

Each is described in more detail below.

4.4.3 Completion of all of a number of communications

In this case the routines test for the completion of *all* of the specified communications (see Figure 16:).

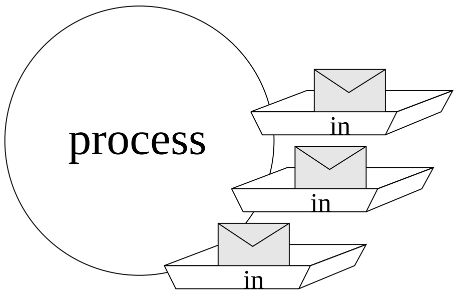

Figure 16: Test to see if all of the communications have completed.

The blocking test is as follows:

```
MPI_WAITALL (count, array_of_requests, array_of_statuses)
```

This routine blocks until all the communications specified by the request handles, `array_of_requests`, have completed. The statuses of the communications are returned in the array `array_of_statuses` and each can be queried in the usual way for the `source` and `tag` if required (see “Information about each message: the Communication Envelope” on page 19“.

There is also a `TEST`-type version which tests each request handle without blocking.

```
MPI_TESTALL (count, array_of_requests, flag, array_of_statuses)
```

If all the communications have completed, `flag` is set to `TRUE`, and information about each of the communications is returned in `array_of_statuses`. Otherwise `flag` is set to `FALSE` and `array_of_statuses` is undefined.

4.4.4 Completion of any of a number of communications

It is often convenient to be able to query a number of communications at a time to find out if any of them have completed (see Figure 17:).

This can be done in MPI as follows:

```
MPI_WAITANY (count, array_of_requests, index, status)
```

`MPI_WAITANY` blocks until one or more of the communications associated with the array of request handles, `array_of_requests`, has completed. The index of the completed communication in the `array_of_requests` handles is returned in `index`, and its status is returned in `status`. Should more than one communication have completed, the choice of which is returned is arbitrary. It is also possible to query if any of the communications have completed without blocking.

```
MPI_TESTANY (count, array_of_requests, index, flag, status)
```

The result of the test (`TRUE` or `FALSE`) is returned immediately in `flag.` Otherwise behaviour is as for `MPI_WAITANY`.

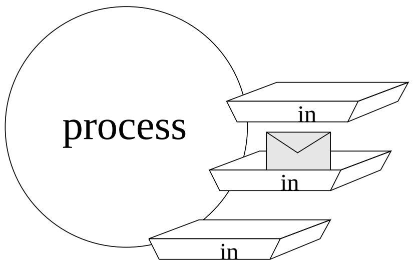

Figure 17: Test to see if any of the communications have completed.

4.4.5 Completion of some of a number of communications

The `MPI_WAITSOME` and `MPI_TESTSOME` routines are similar to the `MPI_WAITANY` and `MPI_TESTANY` routines, except that behaviour is different if more than one communication can complete. In that case `MPI_WAITANY` or `MPI_TESTANY` select a communication arbitrarily from those which can complete, and returns `status` on that. `MPI_WAITSOME` or `MPI_TESTSOME`, on the other hand, return `status` on all communications which can be completed. They can be used to determine how many communications completed. It is not possible for a matched send/receive pair to remain indefinitely pending during repeated calls to `MPI_WAITSOME` or `MPI_TESTSOME` i.e. the routines obey a *fairness* rule to help prevent “starvation”.

```
MPI_TESTSOME (count, array_of_requests, outcount,
array_of_indices, array_of_statuses)
```

4.4.6 Notes on completion test routines

Completion tests deallocate the `request` object for any non-blocking communications they return as complete[1]. The corresponding handle is set to `MPI_REQUEST_NULL`. Therefore, in usual circumstances the programmer would take care not to make a completion test on this handle again. If a `MPI_REQUEST_NULL` request is passed to a completion test routine, behaviour is defined but the rules are complex.

4.5 Exercise: Rotating information around a ring.

Consider a set of processes arranged in a ring as shown below.

Each processor stores its rank in `MPI_COMM_WORLD` in an integer and sends this value onto the processor on its right. The processors continue passing on the values they receive until they get their own rank back. Each process should finish by printing out the sum of the values.

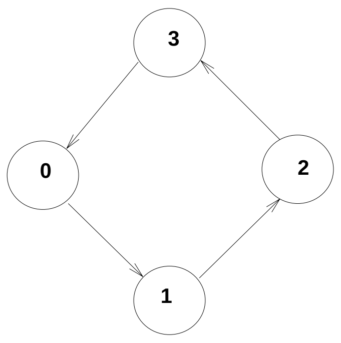

Figure 18: Four processors arranged in a ring.

Extra exercises

1. Modify your program to experiment with the various communication modes and the blocking and non-blocking forms of point-to-point communications.

2. Modify the above program in order to estimate the time taken by a message to travel between to adjacent processes along the ring. What happens to your timings when you vary the number of processes in the ring? Do the new timings agree with those you made with the ping-pong program?

1. Completion tests are also used to test persistent communication requests — see "Persistent communications" on page 66— but do not deallocate in that case.

5 Introduction to Derived Datatypes

5.1 Motivation for derived datatypes

In “Datatype-matching rules” on page 17, the basic MPI datatypes were discussed. These allow the MPI programmer to send messages consisting of an array of variables of the same type. However, consider the following examples.

5.1.1 Examples in C

5.1.1.1 Sub-block of a matrix

Consider

```
double results[IMAX][JMAX];
```

where we want to send `results[0][5]`, `results[1][5]`, `....`, `results[IMAX][5]`. The data to be sent does not lie in one contiguous area of memory and so cannot be sent as a single message using a basic datatype. It is however made up of elements of a single type and is *strided* i.e. the blocks of data are regularly spaced in memory.

5.1.1.2 A struct

Consider

```
struct {
   int nResults;
   double results[RMAX];
} resultPacket;
```

where it is required to send `resultPacket`. In this case the data is guaranteed to be contiguous in memory, but it is of mixed type.

5.1.1.3 A set of general variables

Consider

```
int nResults, n, m;
double results[RMAX];
```

where it is required to send `nResults` followed by `results`.

5.1.2 Examples in Fortran

5.1.2.1 Sub-block of a matrix

Consider

```
DOUBLE PRECISION results(IMAX, JMAX)
```

where we want to send `results(5,1)`, `results(5,2)`, `....`, `results(5,JMAX)`. The data to be sent does not lie in one contiguous area of memory and so cannot be sent as a single message using a basic datatype. It is however made up of elements of a single type and is *strided* i.e. the blocks of data are regularly spaced in memory.

5.1.2.2 A common block

Consider

```
INTEGER nResults
DOUBLE PRECISION results(RMAX)
COMMON / resultPacket / nResults, results
```

where it is required to send `resultPacket`. In this case the data is guaranteed to be contiguous in memory, but it is of mixed type.

5.1.2.3 A set of general variable

Consider

```
INTEGER nResults, n, m
DOUBLE PRECISION results(RMAX)
```

where it is required to send `nResults` followed by `results`.

5.1.3 Discussion of examples

If the programmer needs to send non-contiguous data of a single type, he or she might consider

- making consecutive MPI calls to send and receive each data element in turn, which is slow and clumsy.

  So, for example, one inelegant solution to “Sub-block of a matrix” on page 27, would be to send the elements in the column one at a time. In C this could be done as follows:

```
int count=1;

/*
   **********************************************************
   * Step through column 5 row by row
   **********************************************************
*/

for(i=0;i<IMAX;i++){
   MPI_Send (&(results[i][5]), count, MPI_DOUBLE,
         dest, tag, comm);
}
```

In Fortran:

```
      INTEGER count
C Step through row 5 column by column

      count = 1
```

```
DO i = 1, IMAX
   CALL MPI_SEND (result(i, 5), count, MPI_DOUBLE_PRECISION,
&  dest, tag, comm, ierror)
END DO
```

- copying the data to a buffer before sending it, but this is wasteful of memory and long-winded.

If the programmer needs to send contiguous data of mixed types, he or she might consider

- again, making consecutive MPI calls to send and receive each data element in turn, which is clumsy and likely to be slower.
- using `MPI_BYTE` and `sizeof` to get round the datatype-matching rules, but this produces an MPI program which may not be portable to a heterogeneous machine.

Non-contiguous data of mixed types presents a combination of both of the problems above. The idea of derived MPI datatypes is to provide a portable and efficient way of communicating non-contiguous and / or mixed types in a message.

5.2 Creating a derived datatype

Derived datatypes are created at run-time. Before a derived datatype can be used in a communication, the program must create it. This is done in two stages.

- **Construct the datatype.** New datatype definitions are built up from existing datatypes (either derived or basic) using a call, or a recursive series of calls, to the following routines: `MPI_TYPE_CONTIGUOUS`, `MPI_TYPE_VECTOR`, `MPI_TYPE_HVECTOR`, `MPI_TYPE_INDEXED` `MPI_TYPE_HINDEXED`, `MPI_TYPE_STRUCT`.
- **Commit the datatype.**The new datatype is "committed" with a call to `MPI_TYPE_COMMIT`. It can then be used in any number of communications. The form of `MPI_TYPE_COMMIT` is:

```
MPI_TYPE_COMMIT (datatype)
```

Finally, there is a complementary routine to `MPI_TYPE_COMMIT`, namely `MPI_TYPE_FREE`, which marks a datatype for de-allocation.

```
MPI_TYPE_FREE (datatype)
```

Any datatypes derived from `datatype` are unaffected when it is freed, as are any communications which are using the datatype at the time of freeing. `datatype` is returned as `MPI_DATATYPE_NULL`.

5.2.1 Construction of derived datatypes

Any datatype is specified by its *type map*, that is a list of the form:

basic datatype 0	displacement of datatype 0
basic datatype 1	displacement of datatype 1
...	...
basic datatype *n*-1	displacement of datatype *n*-1

The displacements may be positive, zero or negative, and when a communication call is made with the datatype, these displacements are taken as offsets from the start of the communication buffer, i.e. they are added to the specified buffer address, in order to determine the addresses of the data elements to be sent. A derived datatype can therefore be thought of as a kind of *stencil* laid over memory.

Of all the datatype-construction routines, this course will describe only `MPI_TYPE_VECTOR` and `MPI_TYPE_STRUCT`. The others are broadly similar and the interested programmer is referred to the MPI document [1].

5.2.1.1 MPI_TYPE_VECTOR

```
MPI_TYPE_VECTOR (count, blocklength, stride, oldtype, newtype)
```

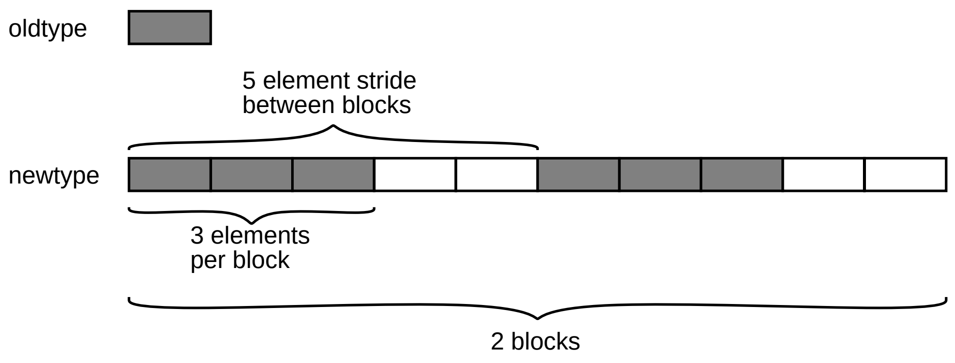

Figure 19: Illustration of a call to `MPI_TYPE_VECTOR` *with* `count = 2`, `stride = 5` *and* `blocklength = 3`

The new datatype `newtype` consists of `count` blocks, where each block consists of `blocklength` copies of `oldtype`. The elements within each block have contiguous displacements, but the displacement between every block is `stride`. This is illustrated in Figure 19:.

5.2.1.2 MPI_TYPE_STRUCT

```
MPI_TYPE_STRUCT (COUNT, ARRAY_OF_BLOCKLENGTHS,
ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES, NEWTYPE)
```

The new datatype `newtype` consists of a list of `count` blocks, where the *i*th block in the list consists of `array_of_blocklengths[i]` copies of the type `array_of_types[i]`. The displacement of the *i*th block is in units of *bytes* and is given by `array_of_displacements[i]`. This is illustrated in Figure 20:.

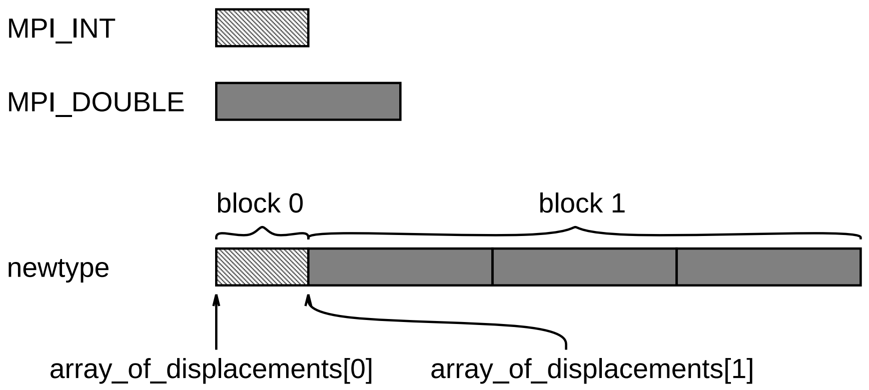

Figure 20: Illustration of a call to `MPI_TYPE_STRUCT` *with* `count = 2`, `array_of_blocklengths[0] = 1`, `array_of_types[0] = MPI_INT`,

`array_of_blocklengths[1] = 3` *and* `array_of_types[1] = MPI_DOUBLE`

See also `MPI_TYPE_SIZE`, `MPI_TYPE_EXTENT`, `MPI_TYPE_LB`, `MPI_TYPE_UB`, `MPI_TYPE_COUNT`

5.3 Matching rule for derived datatypes

A send and receive are correctly matched if the type maps of the specified datatypes, with the displacements ignored, match according to the usual matching rules for basic datatypes. A received message may not fill the specified buffer. The number of *basic* elements received can be retrieved from the communication envelope using `MPI_GET_ELEMENTS`. The `MPI_GET_COUNT` routine introduced earlier returns as usual the number of received elements of the datatype specified in the receive call. This may not be a whole number, in which case `MPI_GET_COUNT` will return `MPI_UNDEFINED`.

5.4 Example Use of Derived Datatypes in C

5.4.1 Sub-block of a matrix (strided non-contiguous data of a single type)

```
double results[IMAX][JMAX];

/* ********************************************************** *
* We want to send results[0][5], results[1][5],
* results[2][5], ...., results[IMAX-1][5]
* ********************************************************** */

MPI_Datatype newtype;

/* ********************************************************** *
* Construct a strided vector type and commit.
* IMAX blocks, each of length 1 element, separated by
* stride JMAX elements * oldtype=MPI_DOUBLE
* ********************************************************** */

MPI_Type_vector (IMAX, 1, JMAX, MPI_DOUBLE,
&newtype);MPI_Type_Commit (&newtype);

/* ********************************************************** *
* Use new type to send data, count=1
* ********************************************************** */

MPI_Ssend(&(results[0][5]), 1, newtype, dest, tag, comm);
```

5.4.2 A C `struct` (data of mixed type)

```
struct{
   int nResults;
   double results[RMAX];
} resultPacket;

/* *********************************************************** *
*     We wish to send resultPacket
* *********************************************************** */

/* *********************************************************** *
* Set up the description of the struct prior to
* constructing a new type.
* Note that all the following variables are constants
* and depend only on the format of the struct. They
* could be declared 'const'.
* ************************************************************ */

#define NBLOCKS 2
int array_of_blocklengths[NBLOCKS] = {1, RMAX};

MPI_Aint array_of_displacements[NBLOCKS];
MPI_Datatype array_of_types[NBLOCKS] = {MPI_INT, MPI_DOUBLE};

/* ***********************************************************
* Use the description of the struct to construct a new
* type, and commit.
* ************************************************************ */

MPI_Datatype resultPacketType;
array_of_displacements[0]=0;

MPI_Type_extent (MPI_INT, &extent);
array_of_displacements[1]=extent;

MPI_Type_struct (2,
                 array_of_blocklengths,
                array_of_displacements,
                array_of_types,
                &resultPacketType);

MPI_Type_commit (&resultPacketType);

/* ***********************************************************
* The new datatype can be used to send any number of
* variables of type 'resultPacket'
* ************************************************************ */

count=1;

MPI_Ssend (&resultPacket, count, resultPacketType, dest, tag,
           comm);
```

5.5 Example Use of Derived Datatypes in Fortran

5.5.1 Sub-block of a matrix (strided non-contiguous data of a single type)

```
      IMPLICIT none
      INTEGER newtype, jmax, imax, newtype, ierror
      INTEGER dest, tag, comm

      DOUBLE_PRECISION results(IMAX, JMAX)

C **************************************************
C We want to send results(5,1), results(5,2)
C results(5,3), ....., results(5, JMAX)
C **************************************************

C **************************************************
C Construct a strided datatype and commit.JMAX blocks,
c each of length 1 element, separated by stride IMAX
c elements.
c
C The old datatype is MPI_DOUBLE_PRECISION
c The new datatype is newtype.
C ***************************************************

      CALL MPI_TYPE_VECTOR (JMAX, 1, IMAX,
   &       MPI_DOUBLE_PRECISION, newtype, ierror)

      CALL MPI_TYPE_COMMIT (newtype, ierror)

C ***************************************************
C Use newtype to send data, count = 1
C ***************************************************

     CALL MPI_SSEND (results(5, 1), 1, newtype, dest,
   &                            tag, comm, ierror)
   .
   .
```

5.5.2 A Fortran common block (data of mixed type)

```
      IMPLICIT none
      INTEGER NBLOCKS
      PARAMETER (NBLOCKS = 2)
      INTEGER nResults, count
      INTEGER array_of_blocklengths(NBLOCKS),
      INTEGER array_of_displacements(NBLOCKS)
      INTEGER array_of_types(NBLOCKS)
      INTEGER array_of_addresses(NBLOCKS)

      DOUBLE PRECISION results(RMAX)
      PARAMETER (RMAX=3)

      COMMON / resultPacket / nResults, results

C ***************************************************
C We want to send resultPacket
C ***************************************************

C ***************************************************
C Set up the description of the common block prior
C to constructing a new type.
C Note that all the following variables are constants
C and depend only on the format of the common block.
C ***************************************************

     array_of_blocklengths(1) = 1
     array_of_blocklengths(2) = RMAX

      CALL MPI_ADDRESS(nResults, array_of_addresses(1), ierror)
      CALL MPI_ADDRESS(results, array_of_addresses(2), ierror)

     array_of_displacements(1) = 0
     array_of_displacements(2) = array_array_of_addresses(2) -
    &                             array_of_addresses(1)

     array_of_types(1) = MPI_INTEGER
     array_of_types(2) = MPI_DOUBLE_PRECISION
```

```
C **************************************************
C Use the description of the struct to construct a
C new type, and commit.
C **************************************************

      CALL MPI_TYPE_STRUCT (NBLOCKS,
                            array_of_blocklengths,
   &                        array_of_displacements,
   &                        array_of_types,
   &                        resultPacketType, ierror)

      CALL MPI_TYPE_COMMIT (resultPacketType, ierror)

C **************************************************
C The new variable can be used to send any number
C of variables of type 'resultPacket'.

C **************************************************

      count = 1

      CALL MPI_SSEND (nResults, count, resultPacketType,
   &                          dest, tag, comm, ierror)
```

5.6 Exercise: Rotating a structure around a ring

Modify the passing-around-a-ring exercise from “Exercise: Rotating information around a ring.” on page 26 so that it uses derived datatypes to pass round either a C structure or a Fortran common block which contains a floating point rankas well as the integer rank. Compute a floting point sum of ranks as well as the integer sum of ranks.

Extra exercises

1. Write a program in which two processes exchange two vectors of the same strided vector data type, e.g. rows or columns of a two-dimensional array. How does the time taken for one message vary as a function of the stride?
2. Modify the above program so that the processes exchange a sub-array of a two-array. How does the time taken for one message vary as a function of the size of the sub-array?

6 Convenient Process Naming: Virtual Topologies

A virtual topology is a mechanism for naming the processes in a communicator in a way that fits the communication pattern better. The main aim of this is to makes subsequent code simpler. It may also provide hints to the run-time system which allow it to optimise the communication or even hint to the loader how to configure the processes — however, any specification for this is outwith the scope of MPI. For example, if your processes will communicate mainly with nearest neighbours after the fashion of a two-dimensional grid (see Figure 21:), you could create a virtual topology to reflect this fact. What this gains you is access to convenient routines which, for example, compute the rank of any process given its coordinates in the grid, taking proper account of boundary conditions i.e. returning `MPI_NULL_PROC` if you go outside the grid. In particular, there are routines to compute the ranks of your nearest neighbours. The rank can then be used as an argument to `MPI_SEND`, `MPI_RECV`, `MPI_SENDRECV` etc. The virtual topology might also gain you some performance benefit, but if we ignore the possibilities for optimization, it should be stressed that nothing complex is going on here: the mapping between process ranks and coordinates in the grid is simply a matter of integer arithmetic and could be implemented simply by the programmer — but virtual topologies may be simpler still.

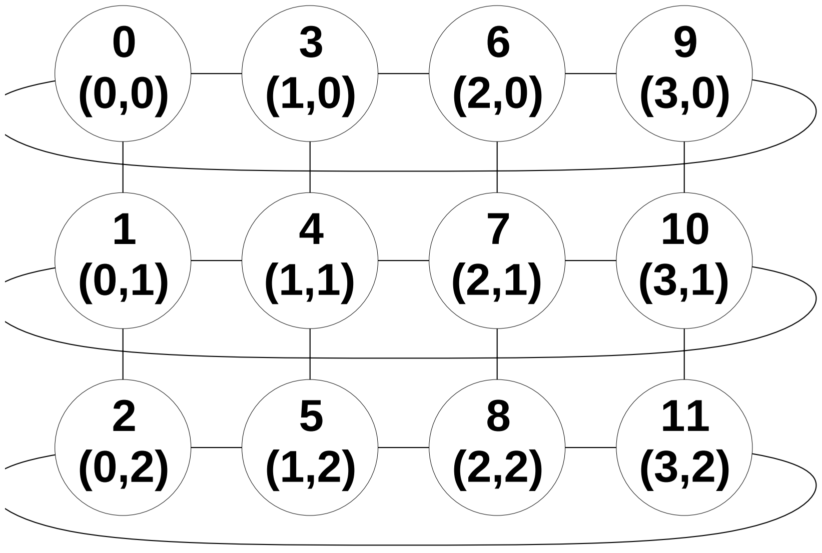

Figure 21: A virtual topology of twelve processes. The lines denote the main communication patterns, namely between neighbours. This grid actually has a cyclic boundary condition in one direction e.g. processes 0 and 9 are ``connected''. The numbers represent the ranks in the new communicator and the conceptual coordinates mapped to the ranks.

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

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

6.1 Cartesian and graph topologies

This course will only describe *cartesian* virtual topologies, suitable for grid-like topologies (with or without cyclic boundaries), in which each process is "connected" to its neighbours in a virtual grid. MPI also allows completely general *graph* virtual topologies, in which a process may be "connected" to any number of other processes and the numbering is arbitrary. These are used in a similar way to cartesian topologies, although of course there is no concept of coordinates. The reader is referred to the MPI document [1] for details.

6.2 Creating a cartesian virtual topology

```
MPI_CART_CREATE (comm_old, ndims, dims, periods, reorder,
comm_cart)
```

`MPI_CART_CREATE` takes an existing communicator `comm_old` and returns a new communicator `comm_cart` with the virtual topology associated with it. The cartesian grid can be of any dimension and may be periodic or not in any dimension, so tori, rings, three-dimensional grids, etc. are all supported. The `ndims` argument contains the number of dimensions. The number of processes in each dimension is specified in the array `dims` and the array `periods` is an array of `TRUE` or `FALSE` values specifying whether that dimension has cyclic boundaries or not. The `reorder` argument is an interesting one. It can be `TRUE` or `FALSE`:

- `FALSE` is the value to use if your data is already distributed to the processes. In this case the process ranks remain exactly as in `old_comm` and what you gain is access to the rank-coordinate mapping functions.
- `TRUE` is the value to use if your data is not yet distributed. In this case it is open to MPI to renumber the process ranks. MPI may choose to match the virtual topology to a physical topology to optimise communication. The new communicator can then be used to scatter the data.

`MPI_CART_CREATE` creates a new communicator and therefore like all communicator-creating routines (see "Communicators, groups and contexts" on page 63) it may (or may not) synchronise the processes involved. The routine `MPI_TOPO_TEST` can be used to test if a virtual topology is already associated with a communicator. If a cartesian topology has been created, it can be queried as to the arguments used to create it (`ndims` etc.) using `MPI_CARTDIM_GET` and `MPI_CART_GET` (see the MPI document [1]).

6.2.1 Note for Fortran Programmers

Fortran programmers should be aware that MPI numbers dimensions from `0` to `ndim - 1`. For example, if the array `dims` contains the number of processes in a particular dimension, then `dims(1)` contains the number of processes in dimension `0` of the grid.

6.3 Cartesian mapping functions

The `MPI_CART_RANK` routine converts process grid coordinates to process rank. It might be used to determine the rank of a particular process whose grid coordinates are known, in order to send a message to it or receive a message from it (but if the process lies in the same row, column, etc. as the calling process, MPI_CART_SHIFT might be more appropriate). If the coordinates are off the grid, the value will be MPI_NULL_PROC for non-periodic dimensions, and will automatically be wrapped correctly for periodic.

```
MPI_CART_RANK (comm, coords, rank)
```

The inverse function MPI_CART_COORDS routine converts process rank to process grid coordinates. It might be used to determine the grid coordinates of a particular process from which a message has just been received.

```
MPI_CART_COORDS (comm, rank, maxdims, coords)
```

The maxdims argument is needed to specify the length of the array coords, usually ndims.

```
MPI_CART_SHIFT (comm, direction, disp, rank_source, rank_dest)
```

This routine does not actually perform a “shift” (see “Shifts and MPI_SENDRECV” on page 67). What it does do is return the correct ranks for a shift which can then be included directly as arguments to MPI_SEND, MPI_RECV, MPI_SENDRECV, etc. to perform the shift. The user specifies the dimension in which the shift should be made in the direction argument (a value between 0 and ndims-1 in both C and Fortran). The displacement disp is the number of process coordinates in that direction in which to shift (a positive or negative number). The routine returns *two* results: rank_source is where the calling process should receive a message *from* during the shift, while rank_dest is the process to send a message *to*. The value will be MPI_NULL_PROC if the respective coordinates are off the grid (see Figure 22: and Figure 23:). Unfortunately, there is no provision for a diagonal “shift”, although MPI_CART_RANK can be used instead.

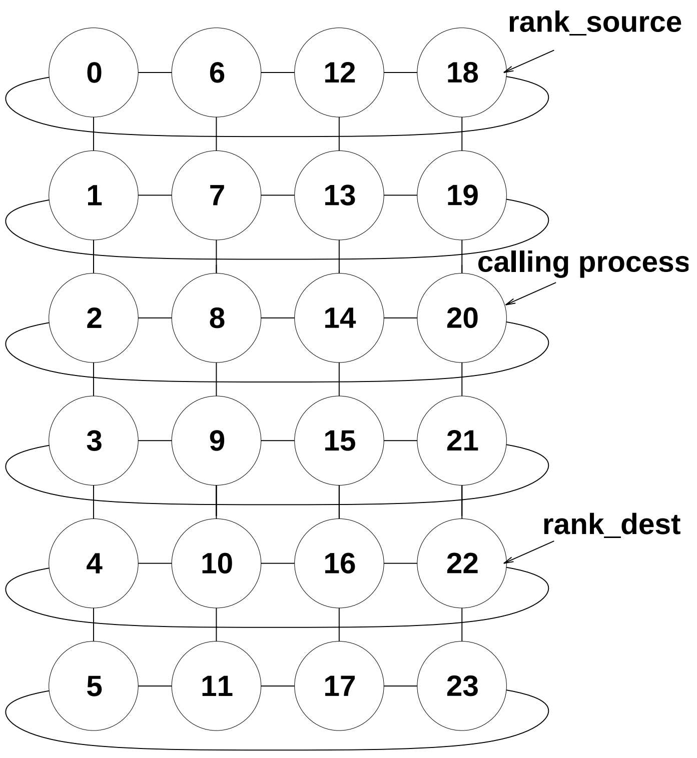

Figure 22: MPI_CART_SHIFT *is called on process 20 with a virtual topology as shown, with* direction=0 *and with* disp=2

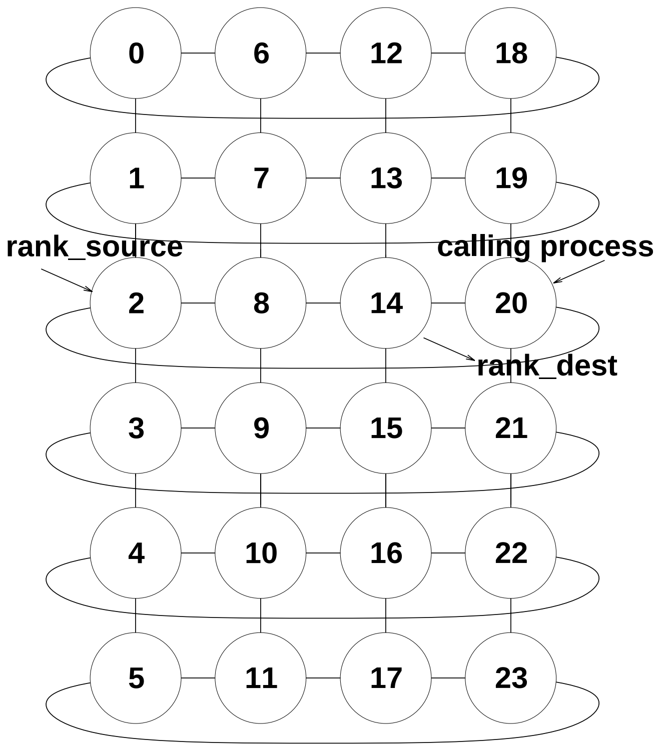

Figure 23: `MPI_CART_SHIFT` *is called on process 20 with a virtual topology as shown, with* `direction=1` *and with* `disp=-1`. *Note the effect of the periodic boundary condition*

6.4 Cartesian partitioning

You can of course use several communicators at once with different virtual topologies in each. Quite often, a program with a cartesian topology may need to perform reduction operations or other collective communications only on rows or columns of the grid rather than the whole grid. `MPI_CART_SUB` exists to create new communicators for sub-grids or “slices” of a grid.

```
MPI_CART_SUB (comm, remain_dims, newcomm)
```

If `comm` defines a 2x3x4 grid, and `remain_dims = (TRUE, FALSE, TRUE)`, then `MPI_CART_SUB(comm, remain_dims, comm_new)` will create three communicators each with eight processes in a 2×4 grid.

Note that only one communicator is returned — this is the communicator which contains the calling process.

6.5 Balanced cartesian distributions

```
MPI_DIMS_CREATE (nnodes, ndims, dims)
```

The `MPI_DIMS_CREATE` function, given a number of processors in `nnodes` and an array `dims` containing some zero values, tries to replace the zeroes with values, to make a grid of the with dimensions as close to each other as possible. Obviously this is not possible if the product of the non-zero array values is not a factor of `nnodes`. This routine may be useful for domain decomposition, although typically the programmer wants to control all these parameters directly.

6.6 Exercise: Rotating information across a cartesian topology

1. Re-write the exercise from page 34 so that it uses a one-dimensional ring topology.
2. Extend one-dimensional ring topology to two-dimensions. Each row of the grid should compute its own separate result.

Extra exercise

Write a program that sorts the rows and columns of a 2-dimensional matrix in increasing order. This is illustrated below with the matrix on the right being the output when the matrix on the left is input. There may be more than one valid output any given input matrix; you need only compute one.

$$\begin{bmatrix} 4 & 0 & 3 \\ 5 & 2 & 7 \\ 2 & 3 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 7 & 5 & 2 \\ 4 & 3 & 1 \\ 3 & 2 & 0 \end{bmatrix}$$

1. In the first instance, assign at most one matrix element to each process.
2. Modify your program so that it can take an arbitrary $N \times N$ matrix for input where N^2 may be much greater than the total number of processes.

7 Collective Communication

MPI provides a variety of routines for distributing and re-distributing data, gathering data, performing global sums etc. This class of routines comprises what are termed the “collective communication” routines, although a better term might be “collective operations”. What distinguishes collective communication from point-to-point communication is that it always involves *every* process in the specified communicator[1] (by which we mean every process in the group associated with the communicator). To perform a collective communication on a subset of the processes in a communicator, a new communicator has to be created (see “When to create a new communicator” on page 64). The characteristics of collective communication are:

- Collective communications cannot interfere with point-to-point communications and *vice versa* — collective and point-to-point communication are transparent to one another. For example, a collective communication cannot be picked up by a point-to-point receive. It is as if each communicator had two sub-communicators, one for point-to-point and one for collective communication.
- A collective communication may or may not synchronise the processes involved[2].
- As usual, completion implies the buffer can be used or re-used. However, there is no such thing as a non-blocking collective communication in MPI.
- All processes in the communicator must call the collective communication. However, some of the routine arguments are not significant for some processes and can be specified as “dummy” values (which makes some of the calls look a little unwieldy!).
- Similarities with point-to-point communication include:
    - A message is an array of one particular datatype (see “What’s in a Message?” on page 7).
    - Datatypes must match between send and receive (see “Datatype-matching rules” on page 17).
- Differences include:
    - There is no concept of tags.
    - The sent message must fill the specified receive buffer.

7.1 Barrier synchronisation

This is the simplest of all the collective operations and involves no data at all.

1.Always an intra-communicator. Collective communication cannot be per formed on an inter-communicator.

2.Obviously `MPI_BARRIER` always synchronises.

```
MPI_BARRIER (COMM)
```

`MPI_BARRIER` blocks the calling process until all other group members have called it.

In one phase of a computation, all processes participate in writing a file. The file is to be used as input data for the next phase of the computation. Therefore no process should proceed to the second phase until all processes have completed phase one.

7.2 Broadcast, scatter, gather, etc.

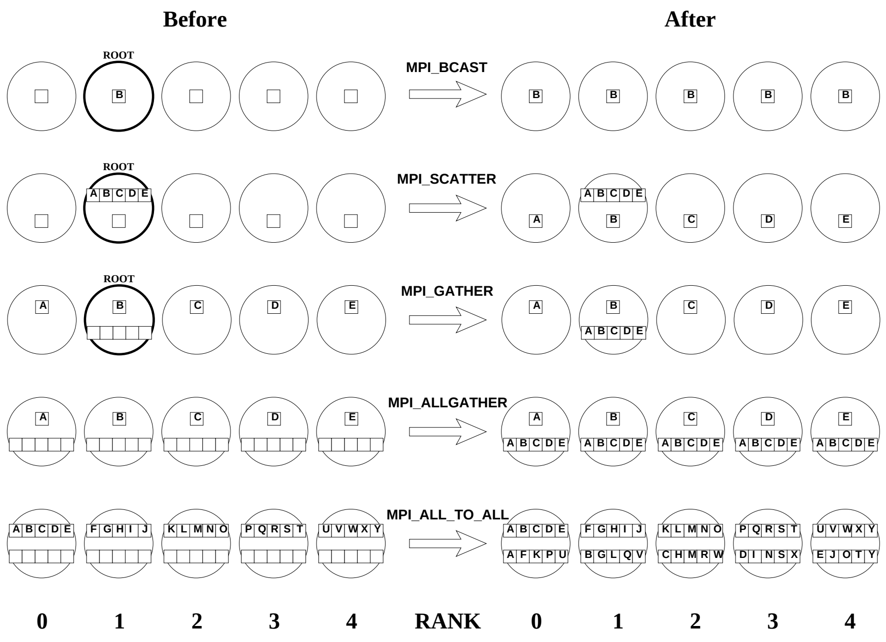

Figure 24: Schematic illustration of broadcast/scatter/gather operations. The circles represent processes with ranks as shown. The small boxes represent buffer space and the letters represent data items. Receive buffers are represented by the empty boxes on the ``before'' side, send buffers by the full boxes.

This set of routines distributes and re-distributes data without performing any operations on the data. The routines are shown schematically in Figure 24:. The full set of routines is as follows, classified here according to the form of the routine call.

7.2.1 `MPI_BCAST`

A broadcast has a specified root process and every process receives one copy of the message from the root. All processes must specify the same root (and communicator).

```
MPI_BCAST (buffer, count, datatype, root, comm)
```

The `root` argument is the rank of the root process. The `buffer`, `count` and `datatype` arguments are treated as in a point-to-point send on the root and as in a point-to-point receive elsewhere.

7.2.2 `MPI_SCATTER`, `MPI_GATHER`

These routines also specify a root process and all processes must specify the same root (and communicator). The main difference from `MPI_BCAST` is that the send and receive details are in general different and so must both be specified in the argument lists. The argument lists are the same for both routines, so only `MPI_SCATTER` is shown here.

```
MPI_SCATTER (sendbuf, sendcount, sendtype, recvbuf, recvcount,
recvtype, root, comm)
```

Note that the sendcount (at the root) is the number of elements to send to *each* process, not to send in total. (Therefore if sendtype = recvtype, sendcount = recvcount). The root argument is the rank of the root process. As expected, for MPI_SCATTER, the sendbuf, sendcount, sendtype arguments are significant only at the root (whilst the same is true for the recvbuf, recvcount, recvtype arguments in MPI_GATHER).

7.2.3 MPI_ALLGATHER, MPI_ALLTOALL

These routines do not have a specified root process. Send and receive details are significant on all processes and can be different, so are both specified in the argument lists. The argument lists are the same for both routines, so only MPI_ALLGATHER is shown here.

```
MPI_ALLGATHER (sendbuf, sendcount, sendtype, recvbuf, recvcount,
recvtype, comm)
```

7.2.4 MPI_SCATTERV, MPI_GATHERV, MPI_ALLGATHERV, MPI_ALLTOALLV

These are augmented versions of the MPI_SCATTER, MPI_GATHER, MPI_ALLGATHER and MPI_ALLTOALL routines respectively. For example, in MPI_SCATTERV, the sendcount argument becomes an array sendcounts, allowing a different number of elements to be sent to each process. Furthermore, a new integer array argument displs is added, which specifies displacements, so that the data to be scattered need not lie contiguously in the root process' memory space. This is useful for sending sub-blocks of arrays, for example, and obviates the need to (for example) create a temporary derived datatype (see"Introduction to Derived Datatypes" on page 27) instead. Full details with examples and diagrams can be found in the MPI document [1].

7.3 Global reduction operations (global sums etc.)

7.3.1 When to use a global reduction operation

You should use global reduction routines when you have to compute a result which involves data distributed across a whole group of processes. For example, if every process holds one integer, global reduction can be used to find the total sum or product, the maximum value or the rank of the process with the maximum value. The user can also define his or her arbitrarily complex operators.

7.3.2 Global reduction operations in MPI

Imagine that we have an operation called "*o*" which takes two elements of an MPI datatype mytype and produces a result of the same type[1].

1.It also has to be associative i.e. A *o (*B *o* C)= (A *o* B) *o* C, meaning that the order of evaluation doesn't matter. The reader should be aware that for floating point operations this is not quite true because of rounding error

Examples include:

1. the sum of two integers
2. the product of two real numbers
3. the maximum of two integers
4. the product of two square matrices
5. a `struct`

   ```
   struct {

      int nResults;

      double results[RMAX];

   } resultPacket;
   ```

   where the operation *o* multiplies the elements in `results` pairwise and sums the `nResults` to produce a result of type `struct resultPacket`
6. a `struct`

   ```
   struct {

      float x;

      int location;

   } fred;
   ```

   where, given two instances of `fred`, $\texttt{fred}_0$ and $\texttt{fred}_1$, the operation `o` compares $\texttt{fred}_0$`.x` with $\texttt{fred}_1$`.x` and sets $\texttt{fred}_{result}$`.x` to the maximum of the two, then sets $\texttt{fred}_{result}$`.location` to be whichever of the two `location`s "won". (A tie is broken by choosing the minimum of $\texttt{fred}_0$`.location` and $\texttt{fred}_1$`.location`.)

A similar thing could be defined in Fortran with an array of two `REAL`s and a bit of trickery which stores the integer `location` in one of the values.

This is in fact the `MPI_MAXLOC` operator (see "Predefined operators" on page 45).

An operation like this can be applied recursively. That is, if we have *n* instances of `mytype` called $\texttt{mydata}_0$ $\texttt{mydata}_1$ ... $\texttt{mydata}_{n-1}$ we can work out[1] $\texttt{mydata}_0$ o $\texttt{mydata}_1$ o ... o $\texttt{mydata}_{n-1}$. It is easiest to explain how reduction works in MPI with a specific example, such as `MPI_REDUCE`.

1.Associativity permits writing this without brackets.

7.3.3 MPI_REDUCE

This is illustrated in Figure 25:.

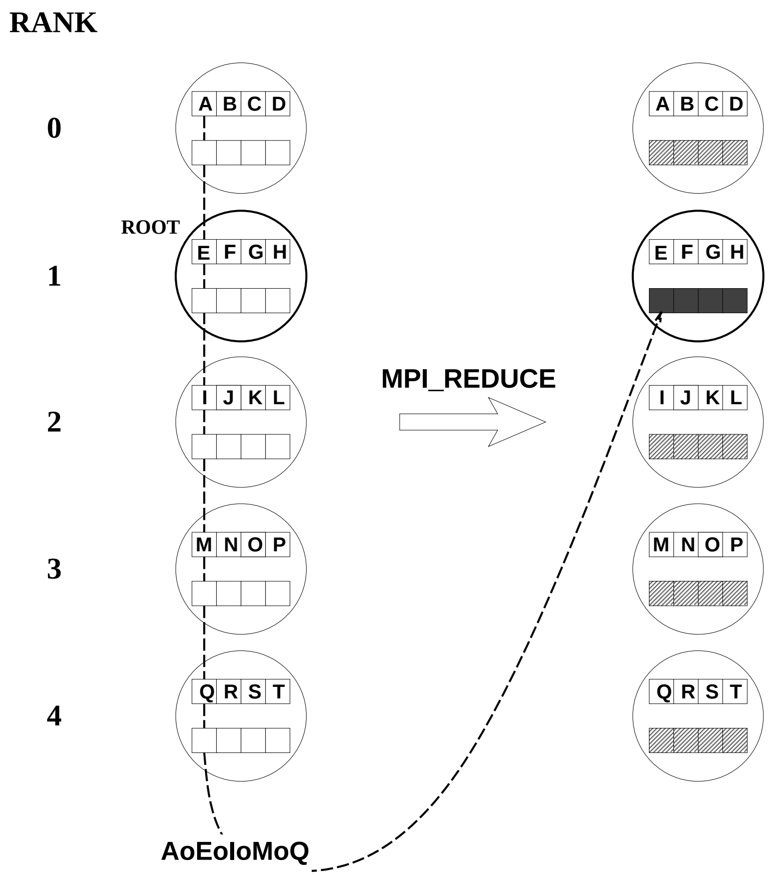

Figure 25: Global reduction in MPI with `MPI_REDUCE`*. o represents the reduction operator. The circles represent processes with ranks as shown. The small boxes represent buffer space and the letters represent data items. After the routine call, the light-shaded boxes represent buffer space with undefined contents, the dark-shaded boxes represent the result on the root. Only one of the four results is illustrated, namely* `A` *o* `E` *o* `I` *o* `M` *o* `Q`*, but the other four are similar --- for example, the next element of the result is* `B` *o* `F` *o* `J` *o* `N` *o* `R`*. Receive buffers are represented by the empty boxes on the ``before'' side, send buffers by the full boxes.*

```
MPI_REDUCE (sendbuf, recvbuf, count, datatype, op, root, comm)
```

All processes in the communicator must call with identical arguments other than `sendbuf` and `recvbuf`. See “Operators” on page 45 for a description of what to specify for the `operator` handle. Note that the root process ends up with an *array* of results — if, for example, a total sum is sought, the root must perform the final summation.

7.3.4 Operators

Reduction operators can be predefined or user-defined. Each operator is only valid for a particular datatype or set of datatypes.

7.3.4.1 Predefined operators

These operators are defined on all the obvious basic C and Fortran datatypes (see Table 7:). The routine `MPI_MAXLOC` (`MPI_MINLOC`) allows both the maximum (minimum) and the rank of the process with the maximum (minimum) to be found. See

"Global reduction operations in MPI" on page 43. More details with examples can be found in the MPI document [1].

Table 7: Predefined operators

MPI Name	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum & location
MPI_MINLOC	Minimum & location

7.3.4.2 User-defined operators

To define his or her own reduction operator, in C the user must write the operator as a function of type `MPI_User_function` which is defined thus:

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

while in Fortran the user must write an `EXTERNAL` subroutine of the following type

```
SUBROUTINE USER_FUNCTION (INVEC(*), INOUTVEC(*), LEN, TYPE)

   <type> INVEC(LEN), INOUTVEC(LEN)

   INTEGER LEN, TYPE
```

The operator must be written schematically like this:

```
for(i = 1 to len)

   inoutvec(i) = inoutvec(i) o invec(i)
```

where *o* is the desired operator. When `MPI_REDUCE` (or another reduction routine is called), the operator function is called on each processor to compute the global result in a cumulative way. Having written a user-defined operator function, it has to be registered with MPI at run-time by calling the `MPI_OP_CREATE` routine.

```
MPI_OP_CREATE (function, commute, op)
```

These return the operator handle `op`, suitable for use in global reduction calls. If the operator is commutative (A *o* B = B *o* A) — the value `commute` should be specified as `TRUE`, as it may allow MPI to perform the reduction faster.

7.3.5 `MPI_ALLREDUCE`, `MPI_REDUCE_SCATTER`, `MPI_SCAN`

These are variants of `MPI_REDUCE`. They are illustrated in Figure 26:,

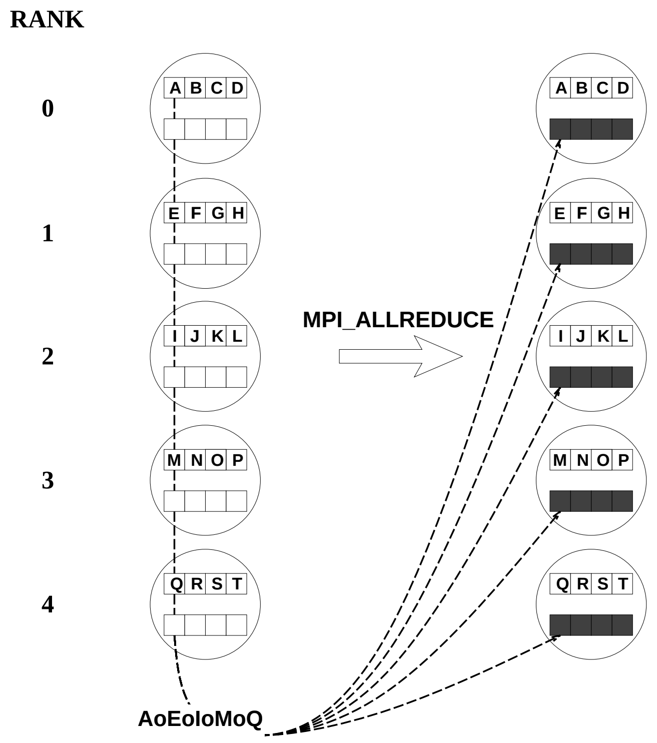

Figure 26: Global reduction in MPI with `MPI_ALLREDUCE`*. The symbols are as in Figure 25:. The only difference from* `MPI_REDUCE` *is that there is no root --- all processes receive the result.*

Figure 27:

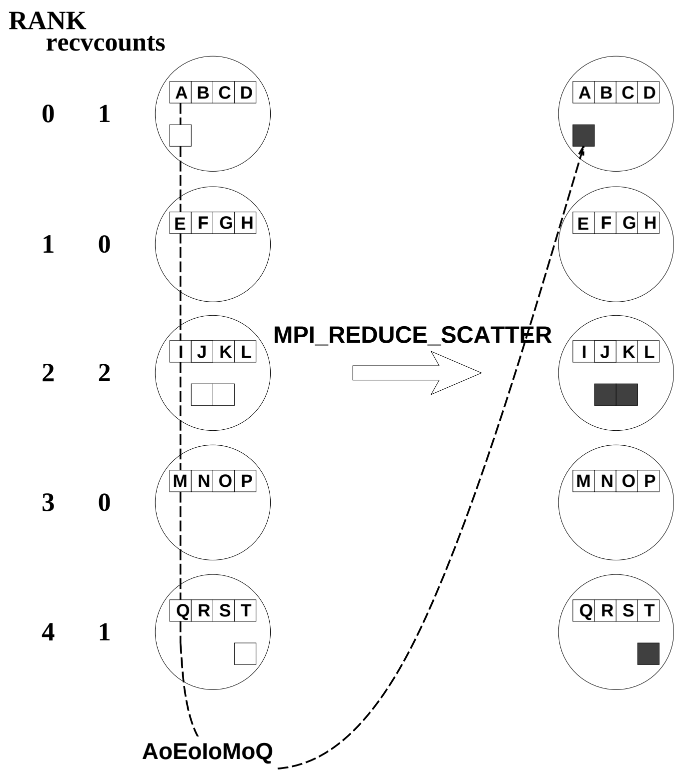

Figure 27: Global reduction in MPI with `MPI_REDUCE_SCATTER`*. The symbols are as in Figure 25:The difference from* `MPI_ALLREDUCE` *is that processes elect to receive a certain-size segment of the result. The segments are always distributed in rank order.*

and Figure 28:.

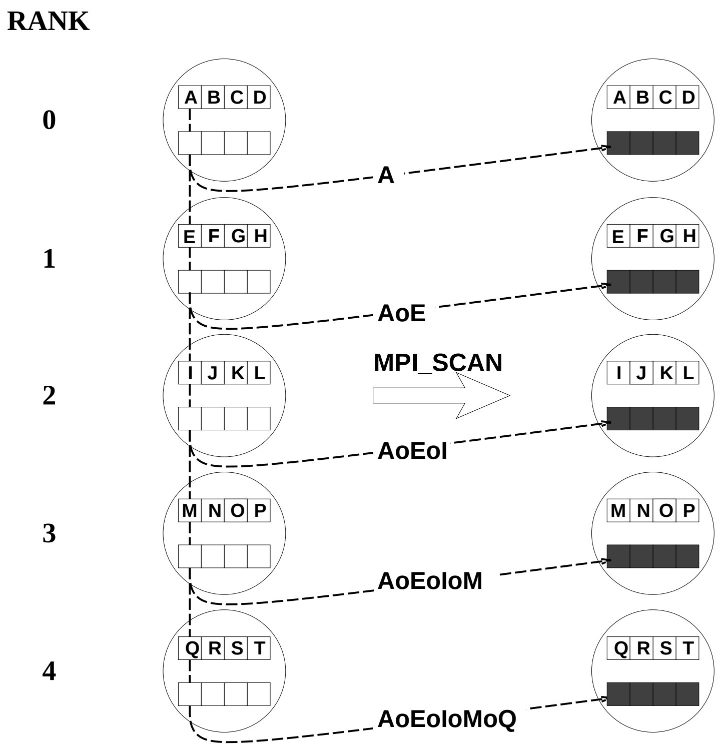

Figure 28: Global reduction in MPI with MPI_SCAN. The symbols are as in Figure 25:. The difference from MPI_ALLREDUCE is that the processes receive a partial result.

The “scan” is sometimes known as a “parallel prefix” operation. Further details of routine arguments and examples (including an implementation of “segmented scan” via a user-defined operator) can be found in the MPI document [1].

7.4 Exercise: Global sums using collective communications

The exercises from Sections 5, 6, and 7 are variations on a global sum where the variable being summed is the ranks of the processors.

1. Re-write the exercise to use MPI global reduction to perform the global sum.
2. Re-write the exercise so that each process prints out a partial sum.
3. Ensure that the processes prints out their partial sum in the correct order, i.e. process 0, then process 1, etc.

Extra exercises

1. Write a program in which the process with rank 0 broadcasts a message via MPI_COMM_WORLD and then waits for every other process to send back that message.
2. Write a program in which the process with rank 0 scatters a message to all processes via MPI_COMM_WORLD and then gathers back that message. How do the execution times of this program and the previous one compare?
3. Define a graph topology where nearest-neighbours of the process with rank k have rank 2k+1 and 2k+2, where $k \geq 0$. Use the resulting communicator to implement a broadcast. How does the execution time of that broadcast vary as a function of the message length? How does it compare with the MPI broadcast?

8 Case Study: Towards Life

8.1 Overview

In this case study you will learn how to:

- Use a master-slave model.
- Perform a domain decomposition and do halo swaps.
- Implement a message passing form of the *Game of Life*.

Each of the above tasks builds on from the previous one. Each is self contained – having completed the previous task – and can be extended using the extra exercises. If you find there is insufficient time to complete the next stage you may care to examine one of the extra exercises instead. Note that these can be performed in any order. If you successfully complete all the steps you should end up with a fully working message passing version of the *Game of Life*. If you do not manage to finish all the steps in the time available don't worry you will still have done something usefu*l*.

8.2 Stage 1: the master slave model

For this part of the exercise you will:

- Create a master slave model – the master will write the output data to file.
- Partition a 2-dimensional array between processors.
- Generate a cartesian virtual topology.
- Processor will colour their data section black or white depending on position.
- Communicate the data back to the master processor which will write it to file in *pgm* format (you are shown how to do this below).

The end result should be a chess-like board, see Figure 29, which can be viewed using the program *xv*.[1]

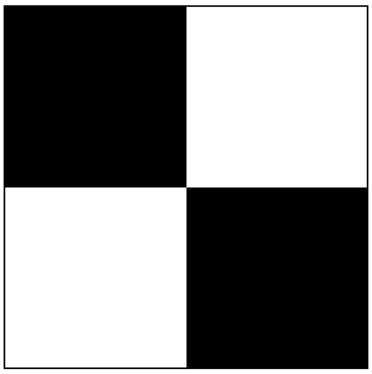

Figure 29: Pattern generated from a 4 processor arrangement.

1. xv is a shareware utility available from: ftp.cis.upenn.edu/pub/xv. See also http://www.trilon.com/xv.

As you will be writing all of the code from scratch *pseudo code* has been provided below. You should easily be able to convert this to C or Fortran code. Note that only the parallel constructs are outlined in the pseudo code.

```
        Initialise MPI

         Find out how many processors there are.
Step 1   set  $n_x$= number of processors in the x direction,
              $n_y$= number of processors in the y direction.
         Create a 2-dimensional, periodic, cartesian grid.

         On processor 0:
              Use a XSIZE by YSIZE integer array.
Step 2        Initialise elements to 0.75*MAXGREY.
         On other processors:
              Use a XSIZE/$n_x$ by YSIZE/$n_y$ integer array.
              Initialize elements to 0.75*MAXGREY.

         Find the cartesian coordinate of the processor: (x,y).
         if((x+y+1)mod2 == 1):
              set elements in local array to 0 (black).
Step 3   else
              set elements in local array to MAXGREY (white).

         Create derived data type(s) to transfer local data
         back to processor 0.

         if processor 0:
              do loop from 1 to number of processors-1
                 receive data using derived data type into
                 the appropriate part of the array.
Step 4        next loop
              write data to file (see below for format).
         else:
              processor sends raw data to processor 0.

        Finalise MPI
```

Step 1

Find the basic info: processor rank and the number of processors used. This step also applies what you have learnt from virtual topologies. The virtual topology will allow you to map the data to the processors and ease the identification of nearest neighbour processors.

It is best not to have fixed values for n_x and n_y. Use the MPI call `MPI_DIMS_CREATE`, see page 180 of the MPI standard, to do the processor assignment.

MPI Routines required: MPI_COMM_SIZE, MPI_COMM_RANK, MPI_DIMS_CREATE, MPI_CART_CREATE.

Step 2

Initially set `XSIZE=YSIZE=128` and `MAXGREY=100`. Note that for the above algorithm to work correctly the number of processors in each of the cartesian directions must divide evenly into XSIZE and YSIZE (see extra exercise 1 if you wish to generalise the scheme). Setting the initial arrays to `0.75*MAXGREY` (grey) will help diagnose problems with the transfer of data.

If you are a C programmer you can allocate the global array only on processor 0. The worker processors need only allocate the amount of memory they will require. For this to work properly though you will have to map a 1d array to a 2d array yourself – this is done to ensure that the memory allocated is contiguous (this may not be the case if you try to allocate memory for a 2d array). The method adopted by Fortran programmers may be easier – see below.

In Fortran77 memory cannot be allocated or deallocated. A *scratch array* has to be used instead. Create the global array across all processors but only use that part required by the processor domain, see Figure 30. Note that if you do this you will need to create a derived data type to send this data (you only want to send the block of data being used) and another at the receiving end to insert the data at the correct location on the master processor.

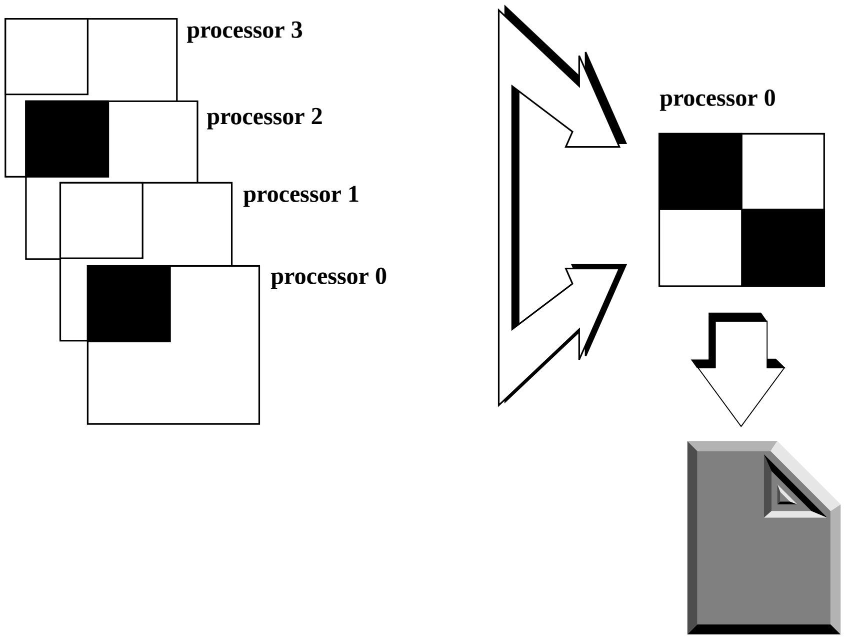

Figure 30: Domain decomposition over four processors. The data is collected back at processor 0 and then written out to file.

Step 3

Once you have determined what portion of the data space a processor is responsible for the data can be initialised. To do this you must know where the processor lies in relation to the global data space. This can be derived from the cartesian coordinate (x,y) in the virtual topology in which the processor lies. If we *paint* processors with (x+y+1) an even number black, white otherwise, the desired chess board pattern will be obtained. If the data initialisation is a little more complex then the process will be a little bit more involved.

MPI Routines s required: MPI_CART_COORDS.

Step 4

The data from the slave processors must be sent back to the master processor which will then reconfigure the global data space and output these numbers to file. The way this process is done is dependent on the way the data is stored on the slave processors.

If the data stored at the slave processors is complete, as opposed to part of a larger data set – say a scratch array, it can just be sent as raw integer data and received at the master processor as a derived data type. This is done to accommodate the received data in the correct positions. If a subset of the data array is used, see Figure 31, a derived data type will have to be used to send the data to the master processor.

To create the derived data types for a data block use `MPI_TYPE_VECTOR`. Remember that in C rows are contiguous in memory while in Fortran it is the columns that are contiguous. Use Figure 31 to help you get the data blocks right.

If you use C the received data should be placed in the global array at element `GArray[dx*coord[0]+coord[1]*dy*XSIZE]` or equivalently `GArray[dx*coord[0]][dy*coord[1]]` in a two dimensional representation. Remember `dx=XSIZE/n`$_x$ and dy=YSIZE/n$_y$ is the amount of data stored locally at each processor.

In Fortran this becomes `Garray(dx*coord(1)+1,dy*coord(2)+1)` – in a one dimensional array representation GArray(dx*coord[0]*XSIZE+coord[1]*dy+1) could be used as in C. Note that if the number of processors do not divide evenly into the extents of the array (*i.e.* columns and/or rows left over) a more complicated process needs to be applied – see extra exercise 1.

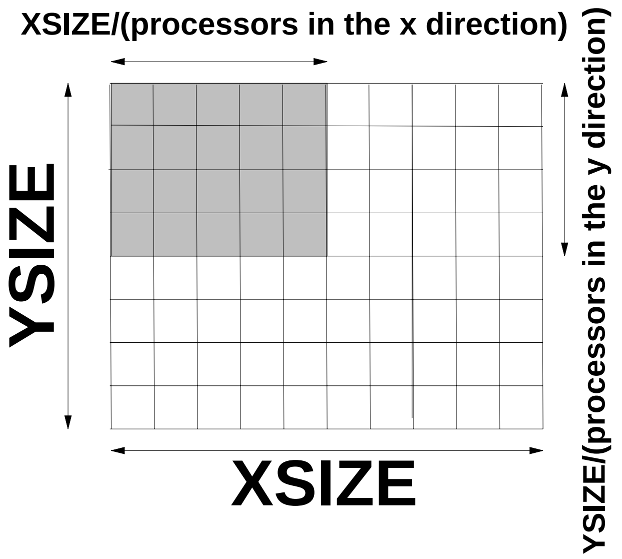
XSIZE/(processors in the x direction)
YSIZE
XSIZE
YSIZE/(processors in the y direction)

Figure 31: Use derived data types to transfer data back to the host processor to output to file.

With this information and the derived data types you should be able to reassemble the data from the distributed domain. The master processor then writes the received data to file in *pgm* (portable grey-map) format. This is quite straightfoward to do. All you need to do is open a file and write in plain text (ASCII) the following:

```
P2
XSIZE YSIZE
MAXGREY
All XSIZE*YSIZE values in the integer array.
```

The P2 at the top of the file is a magic number that identifies the file format. The values XSIZE and YSIZE give the size of the array and MAXGREY gives the maximum contrast – a value of 0 giving black while MAXGREY gives white, intermediate values give different shades of grey. All XSIZE*YSIZE integer values should then be output with no line being more than 70 characters long. Only integers, spaces, tabs and new lines are allowed. More details can be found by typing *man pgm* at the unix prompt.

MPI Routines required: MPI_TYPE_VECTOR, MPI_TYPE_COMMIT, MPI_(I)SEND, MPI_RECV, MPI_WAIT.

If you are successful in this first part of the exercise you should end up with a chess like board as in Figure 29 (when you use 4 processors). Try to make your program general so that it can work on any number of processors (you will have to be careful when the number of processors does not divide evenly into the array dimensions – see extra exercise 1 below). Try different numbers of processors to make sure that the program works. If you have successfully completed this section you can proceed by trying out the extra exercises below or moving on to the next part – boundary swaps.

Extra Exercises

1. Generalise the algorithm to make sure that it can deal with any size of grid and/or number of processors.

There are two ways of doing this: spread the extra rows or columns amongst the first few processors that is if n_x is the number of processors in the x direction and n_y in the y direction then

```
if (XSIZE mod $n_x \neq 0$ AND rank < coord[*]) then  row=row+1
```

and similarly for the columns. Alternatively we can add all the remaining columns or rows to the processor at the edge of the row/column in the virtual topology. This is the approach that we would like you to use and will be expanded in greater detail here – you may still care to employ the other method instead. If the number of columns and/or rows is large enough the load imbalance produced by adopting the latter scheme will be relatively small. You will find that generalising the code however does increase the algorithmic complexity a bit.

A way to incorporate the necessary modification to the algorithm is as follows: if there is a remainder in the number of rows or columns when divided by the number of processors in that direction the processor at the edge of the domain inherits the extra number of columns and/or rows. So:

```
my_rows = XSIZE/nx
xrem = Rem(XSIZE/nx)
if(xrem $\neq$ 0 AND coord[*] = nx- 1) then my_rows = my_rows+xrem
```

where `Rem` denotes the remainder of the expression in brackets. A similar piece of code can be used for the columns. The problem now lies in porting the data back to the host processor.

There are now four possibilities in the message that will be received by the host processor: no extra rows or columns, extra rows and no extra columns, no extra columns but extra rows and finally both extra rows and columns. Derived data types must be created across all processors to take each of these possibilities into account. It is the receiving processors that must use these datatypes to make sure that the data that is sent back to it gets placed in the correct array segment:

```
if(master processor) then
  loop: receive_from = 1 to number of processors - 1
   find out coords of processor receive_from
   if(Rem(XSIZE/$n_x$) $\neq$ 0 AND Rem(YSIZE/$n_y$) $\neq$ 0 AND
        coord[0] == $n_x$-1 AND coord[1] == $n_y$-1) then
            MPI_RECV(...extra_columns_and_rows_data_type....)
    else if(Rem(XSIZE/$n_x$) $\neq$ 0 AND coord[1] == $n_y$-1) then
          MPI_RECV(...extra_columns_data_type....)
    else  if(Rem(YSIZE/$n_y$) $\neq$ 0 AND coord[1] == ny-1) then
          MPI_RECV(...extra_rows_data_type....)
     else
          MPI_RECV(...normal_column_row_data_type...)
     endif
   nextloop
else
    MPI_SEND(my_data,dx*dy,MPI_INTEGER,0,....)
endif
```

The starting position of the incoming data on the array will be the same as before – the derived data types should ensure that everything goes in at the correct place. Once you have done this try it out on different number of processors to make sure that it still works before going on to the next part.

2. Use collective communications to gather the slave processors data back to the master processor.

As you have seen from the course MPI is rather rich in the different types of collective communications available. Unfortunately what is being attempted here cannot be done directly using collective communications as in the general case different derived data types have to be used for the different blocks. We can however perform collective communications over the restricted case where the number of processors divides evenly into the number of rows and columns.

Even with the restriction there is another problem – it is difficult to position the start of the imported data block as the stride in the vector data type will affect the starting location for the data. However there is a trick (but it's a bit of a hack). We can create the column and row derived data types as before but this time this is sandwiched it in a structure data type of integer extent. Using this we can easily position the start and end of the data precisely. A sample piece of C code has been provided below that achieves the object.

```
/* a normal block */
 MPI_Type_vector(dx,dy,YSIZE,MPI_INT,&Block);
 MPI_Type_commit(&MPI_Block);

 /* Now create the structure type */
 offset[0]   = 0;
 MPI_Type_extent(MPI_INT,&extent);
 offset[1]   = extent;
 types[0]    = Block;
 types[1]    = MPI_UB;
 blngths[0]  = 1;
 blngths[1]  = 1;

 MPI_Type_struct(2,blngths,offset,types,&MPI_MBlock);
 MPI_Type_commit(&Block);

 for(i=0;i<size;i++){
      MPI_Cart_coords(GridComm,i,2,coords);
      disp[i]    = dx*coords[0]*YSIZE+dy*coords[1];
      rcounts[i] = 1;
 }

MPI_Gatherv(Larray,dx*dy,MPI_INT,Garray,rcounts,
              disp,Block,0,MPI_COMM_WORLD);
```

The MPI_MBlock data type set an upperbound for the structure using MPI_UB (see §3.12.3 of the MPI Standard, p. 70) which is smaller than the extent of the BLOCK datatype. This allows us to place the correct start of the data using MPI. Mote that in this case the local array only contains dx*dy elements – if the local array is part of a larger data set, as in Figure 30, then you will have to use a derived data type to send the data.

MPI_Gatherv is used to gather the data to the root processor. More information about this call can be found in the MPI standard on p.111. Using this slightly laboured mechanism you can collect all the data in processor 0 using collective communications. As you can see this is not entirely straightforward and anyhow will not work for the general case.

8.3 Stage 2: Boundary Swaps

In this part of the exercise you will:

- Create a halo region around each processor domain.
- Perform halo swaps across processor domains.

In a lot of *domain decomposition*[1] type problems it is often necessary to swap data at the boundaries between processor domains. This is done to minimise the subsequent communication between processors. The data imported from other processors is often referred to as the *halo region*.

You will do this for the *chessboard pattern* program you wrote earlier. In theory you would need to start off by making sure that the local data had extra columns and rows at each of the processor boundaries BUT here we will take a slightly different approach to make sure that things are done correctly. Only update the internal regions of each processor subdomain[2] – *i.e.* do not allocate extra memory, just leave the first and last rows and columns unmodified (this will have to be rewritten for the final part of the case study). These regions can act as the halo to which data from the other processors will be imported to. It will be useful to write the received data to file at this early stage in *.pgm* format. If things are being done correctly you should see that each square is surrounded by a grey boundary as in the left-most diagram in Figure 33 when the output file is viewed using *xv*.

Having done this successfully the next stage is to swap the boundaries between processors. An algorithm you could use is outlined in the piece of *pseudo code* below

```
Create a row derived data type[3]
create a column derived data type
Find nearest neighbours in the x-direction
Find nearest neighbours in the y-direction
Swap boundaries with the processor above and below
Swap boundaries with the processor to the left and right
```

It is as simple as that. Note that when we communicate the haloes not only do we include the internal points but also the outer region of the halo. This ensures that the corners from the opposite domains will be included in the data transferred across,. Figure 32 illustrates the transferral of rows across the processor domains – the same operation would have to be performed for the columns.

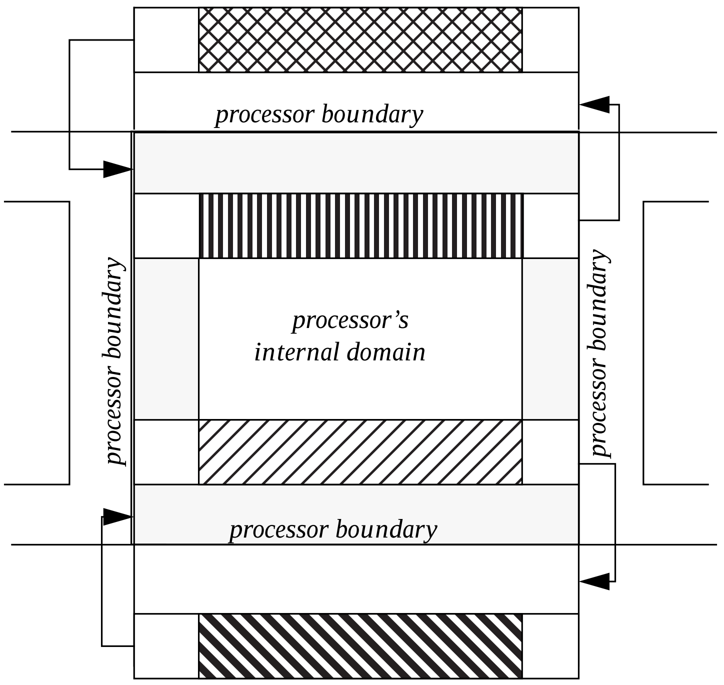

Figure 32: Exchange of rows and columns across processor boundaries.

1. In a domain decomposition the data is distributed amongst the processors as opposed to the functional decomposition method that would attempt to parallelise an algorithm according to the procedures involved.
2. Strictly speaking this is incorrect as you are eating into the data contained within the processor. The halo regions should lie outwith this data. In this particular instance though we are trying to ensure that the data swapping is being done correctly.
3. The extent of these data types should encompass the entire processor subdomain.

Once the data exchange is finished a pattern like the middle diagram in Figure 33 should be obtained. It may be easier to ensure that the operations have been performed correctly by assigning each processor domain an unique shade of grey (see the caption to Figure 33 for an algorithm and the right most diagram for the result) to make sure that data from the correct domains has been exchanged. Inspect the corners carefully to make sure that data from the correct domain has been imported.

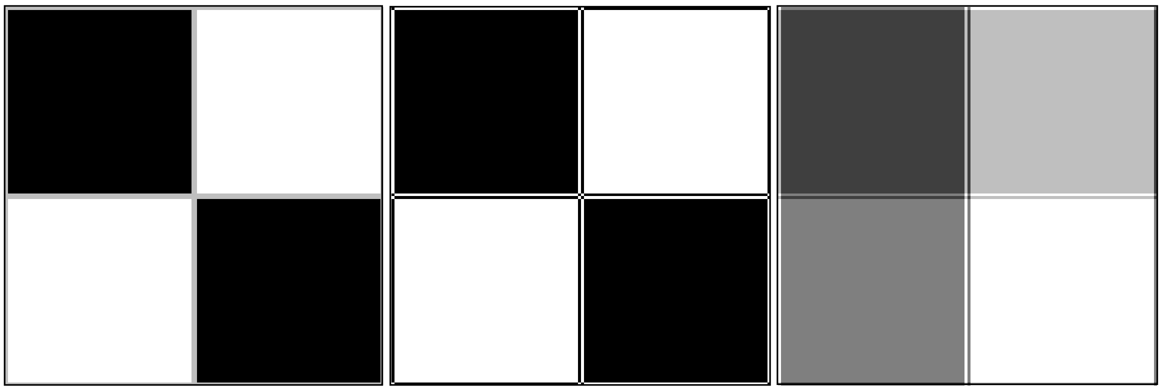

*Figure 33: On left we have the pattern before the boundaries have been swapped – this is a useful check. In the middle the boundaries have been swapped and on the right a greyscale representation has been used. Instead of colouring the blocks black or white they are set to (rank+1)*MAXGREY/(number of processors).*

Now you have all the necessary steps to move on to the next section to construct a full application – the *Game of Life*. If you attend the EPCC HPF course you will be able to see and contrast how the *Game of Life* is constructed using a *data parallel* approach.

8.4 Stage 3: Building the Application

You can now complete the case study by:

- Rewriting the halo routine so that a proper halo region is allocated and only internal regions are updated.
- Derived data types are now used in the slave processors to communicate the internal regions to the master processor excluding the halo.
- Implement the rules of the game of life using the static domain decomposition you have developed in the above exercises.

If you get this far by the end of this exercise you will have implemented a complete application using MPI. The next section briefly describes the Game of Life if you are not familiar with. If you already know this you may wish to skip over it.

What is the Game of Life?

The *Game of Life* is a simple 2-dimensional cellular automata originally conceived by J.H. Conway in 1970. The model evolves a population of organisms in a 2-dimensional space and can exhibit very complex behaviour from a very simple set of evolution rules.

The underlying evolution principle is very simple: cells can be alive or dead at any one time step. The state of the system at the next time step is determined from the number of nearest neighbours each cell has at the present time, see Figure 34. The rules for evolving a system to the next time level are as follows:

- **dead** if the cell has less than two live neighbours – lonely.
- **retain the same state** if the cell has exactly two live neighbours – content.
- **cell is born** if the cell has exactly three live neighbours – ... 8-)
- **die** if the cell has more than three live neighbours – overcrowding.

Using this fairly simple set of rules some fairly complex structures can be exhibited. Figure 37 displays more possible starting configurations you may wish to explore in one of the extra exercises.

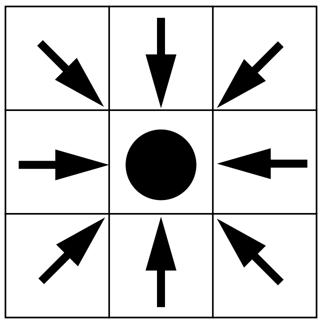

Figure 34: The state of a given cell, alive or dead, is determined from the state of the nearest neighbour cells.

Creating Life

The first thing to do is to make sure that the halo region lies outside the processor's domain data. You will thus now have to allocate (dx+2)x(dy+2) for local data in order to take into account the halo region. Also when communicating data from the slave processors to the master processors derived data types will have to be used to ensure that only the internal region is sent to the master processor. Make sure this is done before you move on to the next part.

For this exercise we will set up a fairly straightforward initial configuration and let it evolve. In this case we have:

$$Cell_{ij} = \begin{array}{ll} Alive & \text{if (i,j)=(XSIZE/2,YSIZE/2)} \\ Dead & \text{otherwise} \end{array}$$

where the i,j subscript refer to the cell's position in the array. The Initial configuration thus consists of a cross across the processor domain. To initialise the data properly we must find a mapping between the local processor's data domains and the global extent of the data as demonstrated in Figure 35.

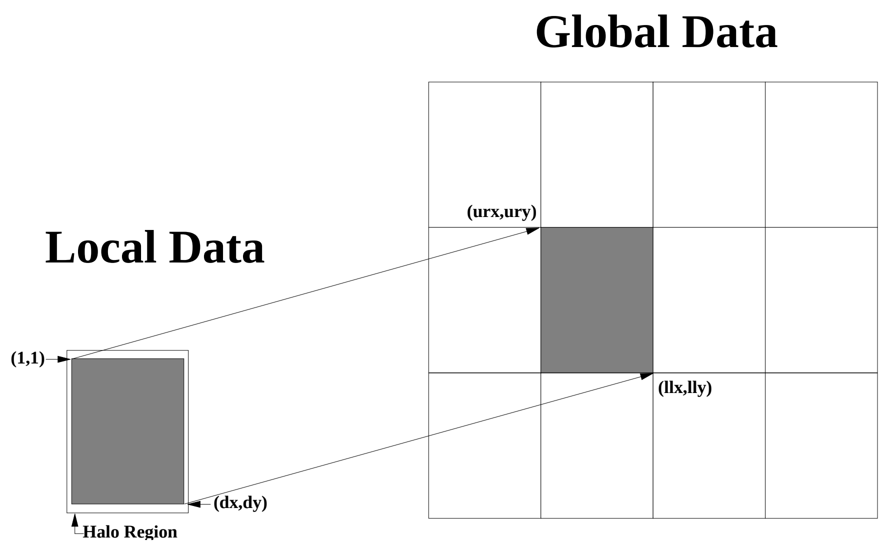

Figure 35: Mapping between the local data coordinates and the global data distribution. NB the local array will have a halo region not shown in the above diagrams.

In order to find a mapping between the local data and the global data all we need to find out is where the upper right hand corner and lower left hand corner of the local data lie in relation to the global data. The mapping is fairly straightforward – make sure you understand it though:

```
urx = 1 + coord[0]*(dx-1)
ury = 1 + coord[1]*(dy-1)
```

and

```
llx = urx + dx
lly = ury + dy
```

These mappings can be used to ensure that the requested initial conditions can be set up locally on each processor. Putting all this together in *pseudo code* we get

```
Set up the Initial Conditions on the Life Board
Loop over the number of iterations
   Swap boundaries
   Clear the count array
   Loop over number of Local Cells
      Count the number of live neighbours cell has
      Store the answer in the Count Array
   next cell
   Loop over number of Cells
      Update the Life board from the Count board
   Next Cell
   Communicate the local Life Board to the Master Processor
   if Master Processor
      Open file and write received data to file
Next iteration
```

Note that two boards are required – one to count the number of nearest live cells and another to keep the current state of the system. Output files should be produced by the master processor at every, or every few iterations, and should be named something like: `life00.pgm`, `life01.pgm`, `life02.pgm`, . . . , `life`*`dd`*`.pgm`. It is now possible to evolve the system according to the rules established for the game of life. To view an animation of the end result use xv as follows:

```
xv -expand 10 -wait 0.5 -wloop -raw life*.pgm
```

to get an animation of your results.

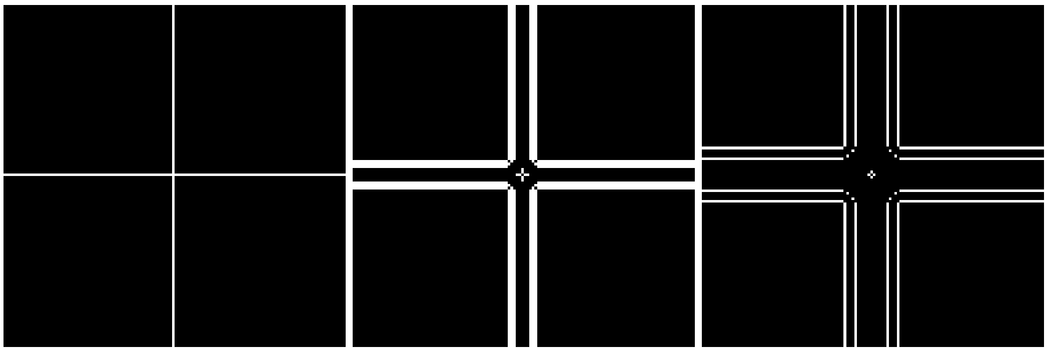

Figure 36: Steps 0, 5 and 10 in the evolution of a 128x128 simulation. Your simulation should develop along the same lines.

Extra Exercises

1. Try to use some of the other initial configurations as given in Figure 37. Explore how the load balance of the system is affected by different configurations. Would a dynamic data decomposition improve matters?

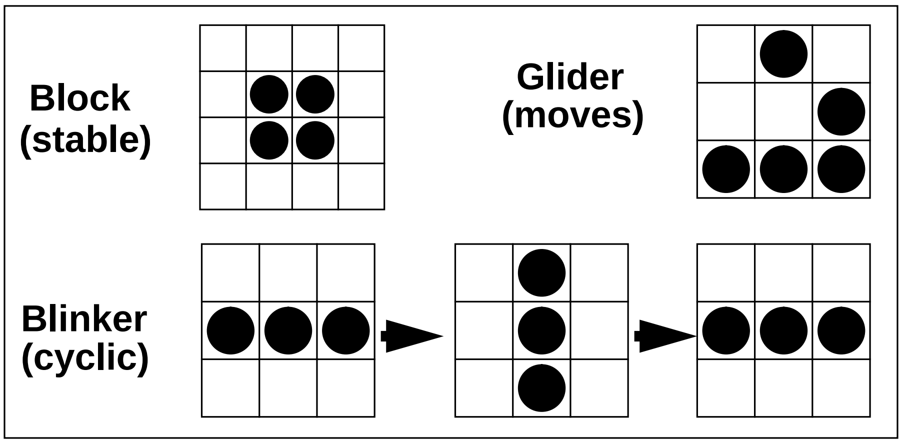

Figure 37: Starting configurations for the game of life. Cells with black circles denote live cells and those with none are dead.

9 Further topics in MPI

9.1 A note on error-handling

A successful MPI routine will always return MPI_SUCCESS, but the behaviour of an MPI routine which detects an error depends on the *error-handler* associated with the communicator involved (or to MPI_COMM_WORLD if no communicator is involved). The following are the two predefined error-handlers associated with MPI_COMM_WORLD[1].

- MPI_ERRORS_ARE_FATAL – This is the default error-handler for MPI_COMM_WORLD. The error is fatal and the program aborts.
- MPI_ERRORS_RETURN – The error causes the routine in which the error occurred to return an error code. The error is not fatal and the program continues executing — however, the state of MPI is undefined and implementation-dependent. The most portable behaviour for a program in these circumstances is to clean up and exit.

The most convenient and flexible option is to register a user-written error-handler for each communicator. When an error occurs on the communicator, the error-handler is called with the error code as one argument. This method saves testing every error code individually in the user's program. The details are described in the MPI document[1] (see MPI_ERRHANDLER_CREATE, MPI_ERRHANDLER_SET, MPI_ERRHANDLER_FREE) but are not discussed in this course.

9.2 Error Messages

MPI provides a routine, MPI_ERROR_STRING, which associates a message with each MPI error code. The format of this routine are as follows:

```
MPI_ERROR_STRING (errorcode, string, resultlen)
```

The array string must be at least MPI_MAX_ERROR_STRING characters long.

9.3 Communicators, groups and contexts

9.3.1 Contexts and communicators

Two important concepts in MPI are those of *communicators* and *contexts*. In fact these two concepts are indivisible, since a communicator is simply the handle to a *context*. Every communicator has a unique context and every context has a unique communicator. A communicator is the central object for communication in MPI. All MPI communication calls require a communicator argument; it follows that all MPI

1.Other communicators, when they are created, inherit error-handlers by default.

communications are made in a specific context. Two MPI processes can only communicate if they share a context and messages sent in one context cannot be received in another. A context is analogous to a radio frequency where only processes which have specified the same frequency can take part in a communication (Figure 38:). Contexts define the scope for communication.

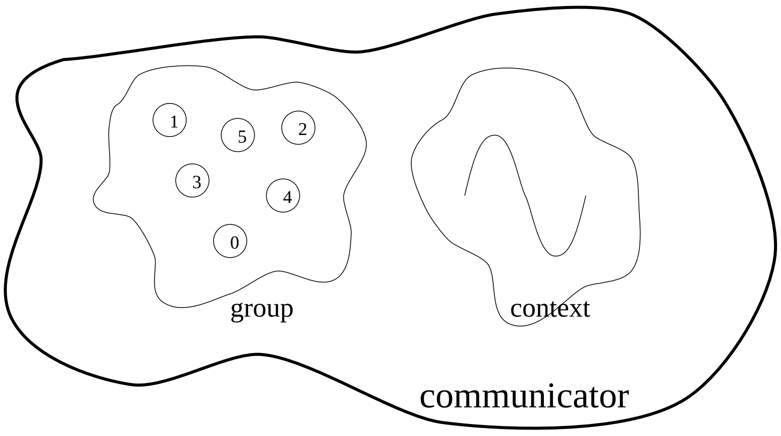

Figure 38: A communicator.

The motivation for context is modularity. The user's code may need to work together with one or more parallel libraries (possibly also written by the same user!), each of which has its own communication patterns. Using context, the communications of each "module" are completely insulated from those of other modules. Note that tags are not suitable for this purpose, since a choice of tags to avoid clashes requires prior knowledge of the tags used by other modules.

9.3.2 When to create a new communicator

It is often the case that a programmer wants to restrict the scope of communication to a subset of the processes. For example:

- The programmer may want to restrict a collective communication to a subset of the processes. For example, a regular domain decomposition may require row-wise or column-wise sums.
- A parallel library may need to re-define the context of user communication to a subset of the original processes (clients) whilst the other processes become servers.

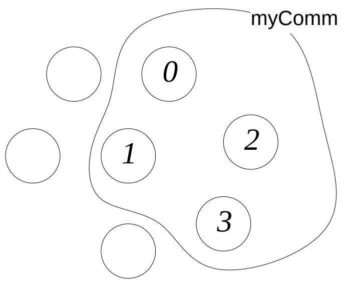

Figure 39: A new communicator defined on a subset of the processes in `MPI_COMM_WORLD`.

There are other reasons for creating a new communicator. When creating a virtual topology (see "Convenient Process Naming: Virtual Topologies" on page 35), a new communicator is automatically created and returned to the user. It simply contains a convenient re-numbering of the group in the original communicator, which typically fits communication patterns better and therefore makes subsequent code simpler.

9.3.3 Communicators and groups

An MPI *group* is simply a list of processes and is *local* to a particular process — processes can create and destroy groups at any time without reference to other processes. Understanding this fact is important in understanding how new communicators are created. It appears to contradict the statement that a communicator / context contains a group, but the point is that *the group contained within a communicator has been previously agreed across the processes at the time when the communicator was set up*, an operation that may synchronise the processes involved.

9.3.4 An aside on intra-communicators and inter-communicators

The “standard” type of communicator is known as an *intra*-communicator, but a second, more exotic type known as an *inter*-communicator also exists[1]to provide communication between two different communicators. The two types differ in two ways:

1. An *intra*-communicator refers to a single group, an *inter*-communicator refers to a pair of groups.The group of an *intra*-communicator is simply the set of all processes which share that communicator.
2. Collective communications (see“Collective Communication” on page 41 can be performed with an *intra*-communicator. They cannot be performed on an *inter*-communicator. The group of processes involved in a collective communication (see “Collective Communication” on page 41) is simply the *group* of the intra-communicator involved.

Inter-communicators are more likely to be used by parallel library designers than application developers. The routines `MPI_COMM_SIZE` and `MPI_COMM_RANK` can be used with *inter*-communicators, but the interpretation of the results returned is slightly different.

9.3.5 The creation of communicators

When a process starts MPI by calling `MPI_INIT`, the single intra-communicator `MPI_COMM_WORLD` is defined for use in subsequent MPI calls. Using `MPI_COMM_WORLD`, every process can communicate with every other. `MPI_COMM_WORLD` can be thought of as the “root” communicator and it provides the fundamental group. New communicators are always created from existing communicators. Creating a new communicators involves two stages:

- The processes which will define the new communicator always share an existing communicator (`MPI_COMM_WORLD` for example). Each process calls MPI routines to form a new group from the group of the existing communicator — these are independent local operations.
- The processes call an MPI routine to create the new communicator. This is a global operation and may synchronise the processes. All the processes have to specify the same group — otherwise the routine will fail.

1.A routine `MPI_COMM_TEST_INTER` exists to query the type of a given communicator.

9.4 Advanced topics on point-to-point communication

9.4.1 Message probing

Message probing allows the MPI programmer to read a communication envelope before choosing whether or not to read the actual message. The envelope contains data on the size of the message and also (useful when wildcards are specified) the source and tag, enabling the programmer to set up buffer space, choose how to receive the message etc. A probed message can then be received in the usual way. This need not be done immediately, but the programmer must bear in mind that:

- In the meantime the probed message might be matched and read by another receive.
- If the receive call specifies wildcards instead of the source and tag from the envelope returned by the probe, it may receive a different message from that which was probed.

The same message may be probed for more than once before it is received. There is one blocking probing routine `MPI_PROBE` and one non-blocking (or “querying”) routine `MPI_IPROBE`. The form of the routines is similar to the normal receive routines — the programmer specifies the source, tag, and communicator as usual, but does not of course specify `buf`, `count` or `datatype` arguments.

```
MPI_PROBE (source, tag, comm, status)
```

`MPI_PROBE` returns when a matching message is “receivable”. The communication envelope `status` can be queried in the usual way, as described in “Information about each message: the Communication Envelope” on page 15.

```
MPI_IPROBE (source, tag, comm, flag, status)
```

`MPI_IPROBE` is similar to `MPI_PROBE`, except that it allows messages to be checked for, rather like checking a mailbox. If a matching message is found, `MPI_IPROBE` returns with `flag` set to `TRUE` and this case is treated just like `MPI_PROBE`. However, if no matching message is found in the “mailbox”, the routine still returns, but with `flag` set to `FALSE`. In this case `status` is of course undefined. `MPI_IPROBE` is useful in cases where other activities can be performed even if no messages of a certain type are forthcoming, in event-driven programming for example.

9.4.2 Persistent communications

If a program is making repeated communication calls with identical argument lists (destination, buffer address etc.), in a loop for example, then re-casting the communication in terms of *persistent communication requests* may permit the MPI implementation to reduce the overhead of repeated calls. Persistent requests are freely compatible with normal point-to-point communication. There is one communication initialisation routine for each send mode (standard, synchronous, buffered, ready) and one for receive. Each routine returns immediately, having created a `request` handle. For example, for standard send:

```
MPI_SEND_INIT (buf, count, datatype, dest, tag, comm, request)
```

The `MPI_BSEND_INIT`, `MPI_SSEND_INIT`, `MPI_RSEND_INIT` and `MPI_RECV_INIT` routines are similar. The `request` from any of these calls can be used to perform communication as many times as required, by making repeated calls to `MPI_START`:

```
MPI_START (request)
```

Each time `MPI_START` is called it initiates a non-blocking instance of the communication specified in the `INIT` call. Completion of each instance is tested with any of the routines described for non-blocking communication in "Testing communications for completion" on page 23. The only difference to the use of the non-blocking communication routines in"Non-Blocking Communication" on page 19 is that completion tests do *not* in this case deallocate the `request` object and it can therefore be re-used. The `request` must be deallocated explicitly with `MPI_REQUEST_FREE` instead.

```
MPI_REQUEST_FREE (request)
```

For example, consider the one-dimensional smoothing example from "Example: one-dimensional smoothing" on page 19 which can be re-written:

```
call MPI_SEND_INIT for each boundary cell;

call MPI_RECV_INIT for each halo cell;

for(iterations) {

   update boundary cells;

   initiate sending of boundary values to neighbours with
MPI_START;

   initiate receipt of halo values from neighbours with MPI_START;

   update non-boundary cells;

   wait for completion of sending of boundary values;

   wait for completion of receipt of halo values;

}

call MPI_REQUEST_FREE to free requests;
```

A variant called `MPI_STARTALL` also exists to activate multiple requests.

9.4.3 Shifts and `MPI_SENDRECV`

A *shift* involves a set of processes passing data to each other in a chain-like fashion (or a circular fashion). Each process sends a maximum of one message and receives a maximum of one message. See Figure 40: for an example. A routine called `MPI_SENDRECV` provides a convenient way of expressing this communication pattern in one routine call without causing deadlock and without the complications of "red-black" methods (see"Motivation for non-blocking communication" on page 20 for a quick description of "red-black").

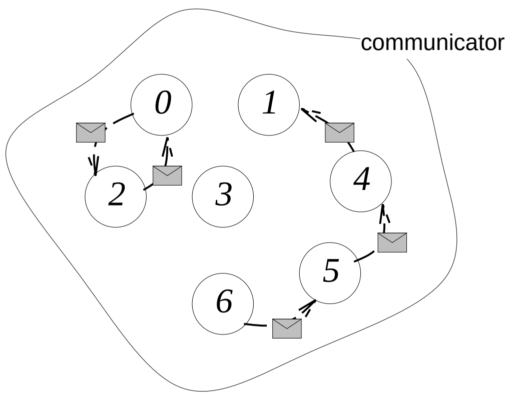

Figure 40: An example of two shifts. `MPI_SENDRECV` *could be used for both*

Note that `MPI_SENDRECV` is just an extension of point-to-point communications. It is completely compatible with point-to-point communications in the sense that messages sent with `MPI_SENDRECV` can be received by a usual point-to-point receive and *vice versa*. In fact, all `MPI_SENDRECV` does is to combine a send and a receive into a single MPI call and make them happen simultaneously to avoid deadlock. It has nothing to do with collective communication and need not involve all processes in the communicator. As one might expect, the arguments to `MPI_SEND_RECV` are basically the union of the arguments to a send and receive call:

```
MPI_SENDRECV (sendbuf, sendcount, sendtype, dest, sendtag,
recvbuf, recvcount, recvtype, source, recvtag, comm, status)
```

There is also a variant called `MPI_SENDRECV_REPLACE` which uses the same buffer for sending and receiving. Both variants are *blocking* — there is no non-blocking form since this would offer nothing over and above two separate non-blocking calls. In figure Figure 40: process 1 only receives, process 6 only sends and process 3 does neither. A nice trick is to use `MPI_NULL_PROC` which makes the code more symmetric. The communication in Figure 40: could work thus with `MPI_SENDRECV`:

Table 8: Communications from Figure 40:

Process	dest	source
0	2	2
1	MPI_NULL_PROC	4
2	0	0
3	MPI_NULL_PROC	MPI_NULL_PROC
4	1	5
6	4	6
6	5	MPI_NULL_PROC

10 For further information on MPI

The first book published about MPI is by Gropp, Lusk and Skjellum and contains a thorough description of MPI and of the major communication interfaces in use when MPI was designed. There is likely to be a flurry of introductory MPI books aimed at specific audiences. Review articles have been written by D. Walker and C.H. Still among others.

Useful Universal Resource Locators:

- Some MPI Home Pages:

  `http://www.mcs.anl.gov/Projects/mpi`

  `ftp://unix.hensa.ac.uk/parallel/standards/mpi`

- MPI Sstandard Document [1]:

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

- MPI Frequently Asked Questions:

  `http://www.cs.msstate.edu/dist_computing/mpi-faq.html`

- EPCC TEC Technical Watch Report:

  `http://www.epcc.ed.ac.uk/epcc-tec`

- EPCC Native Implementation of MPI:

  `http://www.epcc.ed.ac.uk/t3dmpi/Product`

- A list of MPI implementations:

  `http://www.osc.edu/mpi/`

11 References

[1] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, 19945 (PostSCript or HTML versions available at http://www.epcc.ed.ac.uk/epcc-tec/documents/otherres.html)

[2] William Gropp, Ewing Lusk and Anthony Skjellum. “Using MPI: Portable Parallel Programming with the Message Passing" (2nd edition), MIT Press, 1999, ISBN: 0-262-57132-3.

[3] William Gropp, Ewing Lusk and Anthony Skjellum. “Using MPI-2: Advanced Features of the Message Passing Interface, MIT Press, 1999, ISBN: 0-262-57133-1.

[4] Peter S Pacheko. “Parallel Programming with MPI”, Morgan Kaufmann, 1997.

[5] M. Snir, S.W.Otto, S. Huss-Lederman, D. Walker & j. Dongarra,“MPI - The Complete Reference’’ (2 volumes), MIT Press, 1998,

[6] C.H. Still: Portable Parallel Computing Via the MPI1 Message-Passing Standard. Computers in Physics, 8(5), pp533-8, Sept./Oct. 1994.

[7] D. Walker: The design of a standard message-passing interface for distributed memory concurrent computers. Parallel Computing, 20(4), pp 657-73, Apr. 1994.

Appendix A: Compiling and Running MPI Programs on lomond

This appendix contains information on how to compile MPI programs on the University of Edinburgh HPC Service (`lomond`). General information on running programs on this service can be found in the document "*Introduction to the University of Edinburgh HPC Service*" which is available at:

`http://www.epcc.ed.ac.uk/sun/introdoc.html`

A.1 Compilation

A.1.1 Fortran compilation

Fortran source files are compiled using the `tmf90` command

To compile the file `hello.F`, you would type the command:

```
lomond$ tmf90 -o hello hello.F -lmpi
```

You may use extensions `.f` or `.F` (FORTRAN 77 fixed format layout) or `.f90` or `.F90` (Fortran 90 free-format layout).

A.1.2 C compilation

C source files are compiled using the `tmcc` command

To compile the file `hello.c`, you would type the command:

```
lomond$ tmcc -o hello hello.c -lmpi
```

A.2 Execution

To execute a compiled (C or Fortran) program:

```
lomond$ bsub -I -q fe-int -n 2 pam ./hello
```

This enters the job interactively into the `fe-int` queue on 2 processors.

Alternatively:

```
lomond$ bsub -o logfile [-x] -q hpc-course -n 4 pam ./hello
```

enters the job into the `hpc-course` queue on 4 processors, storing the results in a file called `logfile`. Use of the (optional) `-x` switch gives exclusive access to the machine which is useful for timing purposes when necessary.

The job start software `pam` is required for all queues.

A.3 Use of MPI with Fortran 90

Whether compiling by hand or using make files, the user should be aware that there are no Fortran 90 bindings with MPI yet. Programs utilising MPI should use FORTRAN 77 syntax and constructs (although Fortran 90 file layout is permitted). Use of Fortran 90 features such as user-defined data types, or array sections in MPI calls is not allowed.

We stress again the fact that there are no Fortran 90 bindings available with MPI and that extreme care should be taken when using MPI with CF90.

The support for Fortran which is available with this implementation of MPI corresponds more or less to the ``Basic Fortran Support'' described in section 10.2 of the draft MPI-2 standard. This is viewable on the WWW at:

`http://www.epcc.ed.ac.uk/epcc-tec/documents/mpi-20-html/node234.html`

Users intending to use MPI with Fortran 90 should study this information carefully, as there are several issues whose significance must be fully appreciated before MPI can be used with confidence in this context.

For example, array sections must not be passed to to non-blocking operations because of copy-in / out problems. So, the non-blocking:

```
real :: x(8), y(8,8)
   .
call MPI_isend(x(1:4),4,...)
call MPI_isend(y(3,:),8,...)
   .
```

should be avoided, whereas the blocking is allowed:

```
real :: x(8), y(8,8)
   .
call MPI_send(x(1:4),4,...)
call MPI_send(y(3,:),8,...)
   .
```

Here is a very simple example of the use of MPI with CF90:

```
kelvin: cat hello.f90
! prints Hello message, and stops.

program hello
implicit none
include “mpif.h”
integer ierror, rank, size

! initialise mpi
call mpi_init(ierror)

! get ranks (processor number)
call mpi_comm_rank(mpi_comm_world,rank,ierror)
call mpi_comm_size(mpi_comm_world,size,ierror)

! main program
write(unit=6, fmt=*)’Hello from processor ‘,rank,’ of ‘,size,’!’
```

```
! close mpi
call mpi_finalize(ierror)

end

kelvin: f90 -X4 -lmpi -lsma -I/usr/include/mpp hello.f90

darwin: a.out
 Hello from processor  0  of  4 !
 Hello from processor  2  of  4 !
 Hello from processor  3  of  4 !
 Hello from processor  1  of  4 !
darwin:
```

A.4 Using a Makefile

Sometimes – especially if a large number of source files are being used, it is convenient to use a Makefile for compilation. Below two template Makefiles have been provided to compile Fortran or C code. These have been made as simple as possible.

First, for Fortran code we have:

```
#######################################################################
# Fortran sample Makefile.
#######################################################################
# Fortran sources.
SRC= ising.f startup.f sweeps.f energy.f edges.f

OBJ=$(SRC:.f=.o)

FC     = tmf90

# Flags used for compilation:
#            -c     as usual

FFLAGS =

LDFLAGS =
LIBS    = -lmpi

.f.o:
      $(FC) $(FFLAGS) $<

ising: $(OBJ)
      $(FC) $(LDFLAGS) -o $@ $(OBJ) $(LIBS)
```

and for C code we have:

```
#######################################################################
# Example C Makefile.
#######################################################################
# C sources.
SRC    =ising.c startup.c sweeps.c energy.c edges.c

OBJ    =$(SRC:.c=.o)

CC     = tmcc

# -c         as usual
```

```
CFLAGS  = -c
LIBS    = -lmpi

.c.o:
	$(COMP) $(CFLAGS) $<

ising: $(OBJ)
	$(COMP) $(LDFLAGS) -o $@ $(OBJ) $(LIBS)
```