Writing Message-Passing Parallel Programs with MPI

A two-day course

Course Notes

Neil MacDonald, Elspeth Minty, Tim Harding, Simon Brown

Edinburgh Parallel Computing Centre

The University of Edinburgh

Table of Contents

1 Getting Started

1.1 The message-passing programming model

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

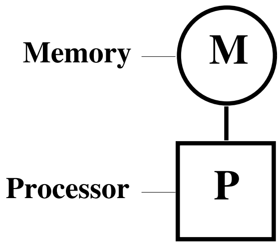

Figure 1: The sequential programming paradigm

The message-passing paradigm is a development of this idea for the purposes of parallel programming. Several instances of the sequential paradigm are considered together. That is, the programmer imagines several processors, each with its own memory space, and writes a program to run on each processor. So far, so good, but parallel programming by definition requires co-operation between the processors to solve a task, which requires some means of communication. The main point of the message-passing paradigm is that the processes communicate by sending each other messages. Thus the message-passing model has no concept of a shared memory space or of processors accessing each other's memory directly — anything other than message-passing is outwith the scope of the paradigm[1]. As far as the programs running on the individual processors are concerned, the message passing operations are just subroutine calls.

1.Readers with experience of data-parallel programming will see how message-passing contrasts with and complements the data-parallel model.

Those with experience of using networks of workstations, client-server systems or even object-oriented programs will recognise the message-passing paradigm as nothing novel.

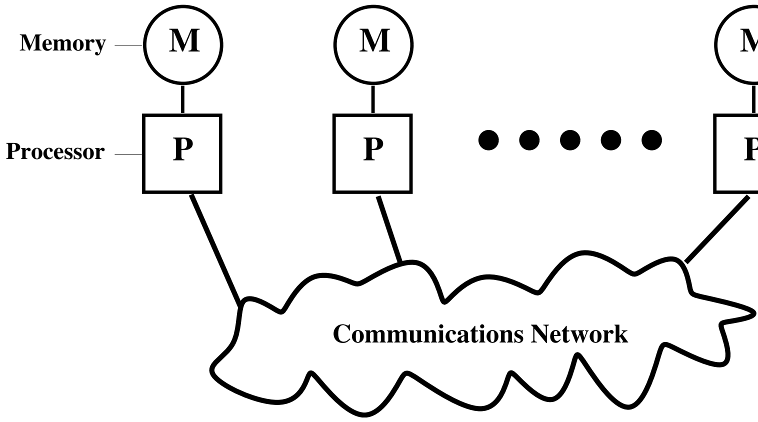

Figure 2: The message-passing programming paradigm.

The message-passing paradigm has become increasingly popular in recent times. One reason for this is the wide number of platforms which can support a message-passing model. Programs written in a message-passing style can run on distributed or shared-memory multi-processors, networks of workstations, or even uni-processor systems. The point of having the paradigm, just as in the sequential case, is that the programmer knows that his or her algorithms should in principle be portable to any architecture that supports a message-passing model[1]. Message-passing is popular, not because it is particularly easy, but because it is so *general*.

1.1.1 What is SPMD?

In the section above, the message-passing paradigm was described as involving a set of sequential programs, one for each processor. In reality, it is rare for a parallel programmer to make full use of this generality and to write a different executable for each processor. Indeed, for most problems this would be perverse — usually a problem can naturally be divided into sub-problems each of which is solved in broadly the same way. An example is the transformation or iterative update of a regular grid (perhaps in image processing or the numerical modelling of a problem from physics). The same operation is to be applied at every grid point. A typical parallel algorithm divides the grid into sub-grids and each sub-grid is processed in the same way.

Typically then, a programmer will want to write one program which is to be replicated across multiple processors (probably as many as possible!), often with a one-off controller process and possibly with other one-off processes like a name-server etc.

The acronym "SPMD" stands for *single-program-multiple-data* and refers to a restriction of the message-passing model which requires that all processes run the same executable. Some vendors provide parallel environments which only support SPMD parallel programs. In practice this is not usually a problem to the programmer, who can incorporate all the different types of process he or she requires into one overall executable. For example, here a `controller` process performs a different task (e.g. reading, checking and distributing initial data) to a `worker` process:

1. To a first approximation: Of course in reality, just as in the sequential case, any programmer who cares about performance (i.e. all of us some of the time) should give some thought to the target architecture.

```
main(int argc, char **argv)

   if(process is to become a controller process)

      Controller( /* Arguments */ );

   else

      Worker( /* Arguments */ );
```

or in Fortran,

```
PROGRAM

IF (process is to become a controller process) THEN

   CALL CONTROLLER( /* Arguments */ )

ELSE

   CALL WORKER( /* Arguments */ )

ENDIF

END
```

Often, for related reasons of efficiency, some vendors do not allow time-sharing i.e. multiple processes per processor (some authorities understand the term “SPMD” to include this further restriction). The programmer should bear in mind that in a SPMD environment in which multiple processes per processor are not allowed, having special lightly-loaded one-off processes such as “controllers” or name-servers may be inefficient because a whole processor will be taken up with that process.

1.2 Messages

A message transfer is when data moves from variables in one sub-program to variables in another sub-program. The message consists of the data being sent. The message passing system has no interest in the value of this data. It is only concerned with moving it. In general the following information has to be provided to the message passing system to specify the message transfer.

- Which processor is sending the message.
- Where is the data on the sending processor.
- What kind of data is being sent.
- How much data is there.
- Which processor(s) are receiving the message.
- Where should the data be left on the receiving processor.
- How much data is the receiving processor prepared to accept.

In general the sending and receiving processors will cooperate in providing this information. Some of this information provided by the sending processor will be attached to the message as it travels through the system and the message passing system may make some of this information available to the receiving processor.

As well as delivering data the message passing system has to provide some information about progress of communications. A receiving processor will be unable to use incoming data if it is unaware of its arrival. Similarly a sending processor may wish to find out if its message has been delivered. A message transfer therefore provides synchronisation information in addition to the data in the message.

The essence of message passing is communication and many of the important concepts can be understood by analogy with the methods that people use to communicate, phone, fax, letter, radio etc. Just as phones and radio provide different kinds of service different message passing systems can also take very different approaches. For the time being we are only interested in general concepts rather than the details of particular implementations.

1.3 Access

Before messages can be sent a sub-program needs to be connected to the message passing system. This is like having a telephone installed or a a mailbox fitted to the front door. A person with two telephones can use one phone for business use and one phone for personal calls. Some message passing systems also allow a single processor to have multiple connections to the message passing system. Other systems only support a single connection to the network so some other mechanism must be used to distinguish between different types of message.

1.4 Addressing

Messages have to be addressed in some way. Postal mail needs a town, street and house number. Fax messages require a phone number. A single letter may contain several completely independent items. For example my credit card bills invariably comes packaged with a mail order catalogue. However they were contained within a single envelope with a single address and travelled through the mail system as a single item so they count as a single message.

The postal system only looks at the information on the envelope. As well as the address the message envelope may provide additional information, for example a return address or some indication of the contents of the message. It is usually possible to separate the morning mail into bills and letters without opening any of them.

The same is true of most message passing systems. Each message must be addressed. The data is moved through the message passing system association with some ``envelope'' information that includes this address. When the message is received the receiving process may also have access to some of this information.

1.5 Reception

It is important that the receiving process is capable of dealing with the messages that have been sent. If a processor is sent a message than it is in-capable of handling (e.g. larger than the buffer the processor is putting the message into) then various strange effects can occur. For example the message passing system may truncate or discard the message.

These buffers may be variables declared within the application code or they may be internal to the message passing system.

1.6 Point to Point Communication

The simplest form of message is a point to point communication. A message is sent from the sending processor to a receiving processor. Only these two processors need to know anything about the message.

There are several variations on how the sending of a message can interact with the execution of the sub-program.

The first common distinction is between *synchronous* and *asynchronous* sends.

Synchronous sends are provided with information about the completion of the message.

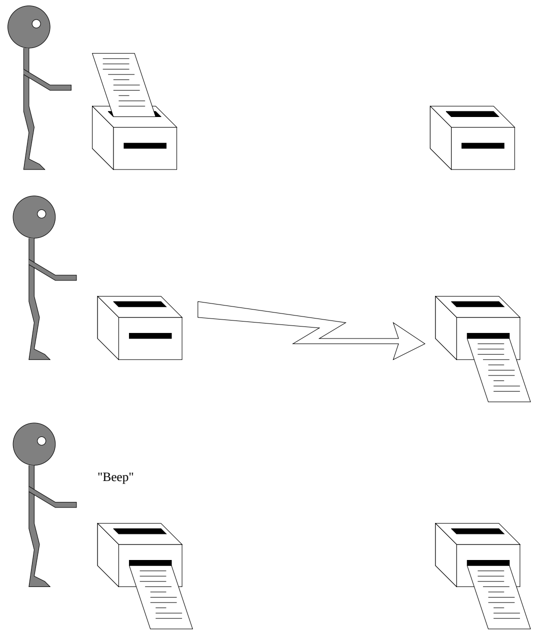

Figure 3: A synchronous communication does not complete until the message has been received.

Asynchronous sends only know when the message has left.

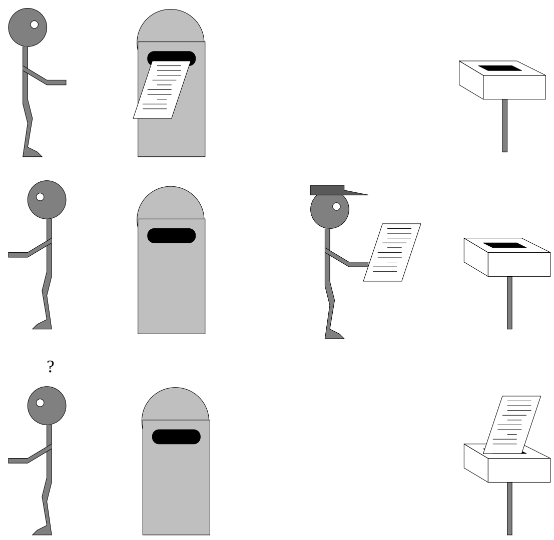

Figure 4: An asynchronous communication completes as soon as the message is on its way.

A fax message or registered mail is a synchronous operation. The sender can find out if the message has been delivered.

A post card is an asynchronous message. The sender only knows that it has been put into the post-box but has no idea if it ever arrives unless the recipient sends a reply.

The other important distinction is *blocking* and *non-blocking*.

Blocking operations only return from the subroutine call when the operation has completed.

Non-blocking operations return straight away and allow the sub-program to continue to perform other work. At some later time the sub-program can test for the completion of the non-blocking operation.

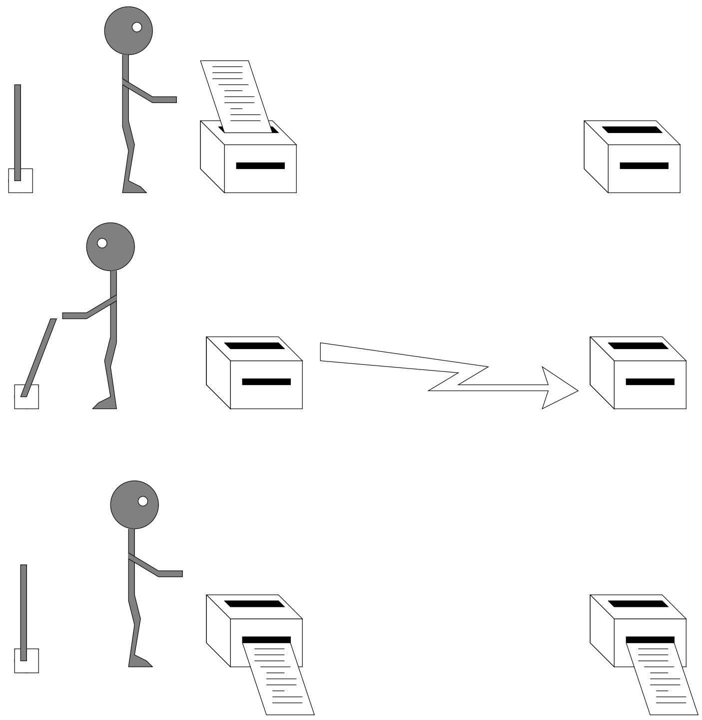

Figure 5: Non blocking communication allows useful work to be performed while waiting for the communication to complete

Normal fax machines provide blocking communication. The fax remains busy until the message has been sent. Some modern fax machines contain a memory. This allows you to load a document into the memory and if the remote number is engaged the machine can be left to keep trying to get through while you go and do something more important. This is a non-blocking operation.

Receiving a message can also be a non-blocking operation. For example turning a fax machine on and leaving it on, so that a message can arrive. You then periodically test it by walking in to the room with the fax to see if a message has arrived.

1.7 Collective Communications

Up until now, we've only considered point-to-point communications those involving a pair of communicating processes. Many message-passing systems also provide operations which allow larger numbers of processes to communicate.

All of these operations can be built out of point to point communications but it is a good idea use provided routines if they exist.

1.7.1 Barrier

A barrier operation synchronises processors. No data is exchanged but the barrier blocks until all of the participating processors have called the barrier routine.

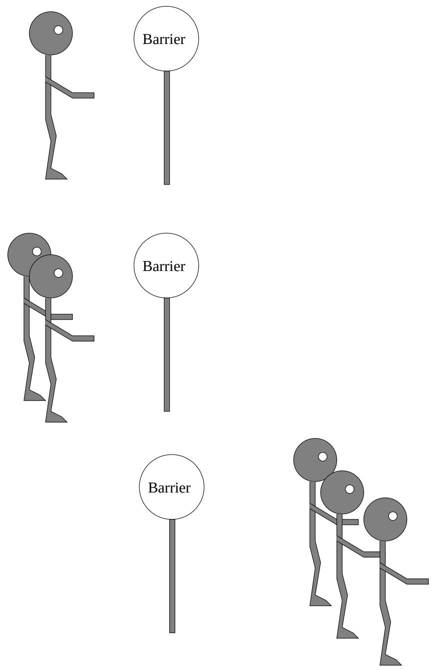

Figure 6: A barrier operation synchronises a number of processors.

1.7.2 Broadcast

A broadcast is a one-to-many communication. One processor send the same message to several destinations with a single operation.

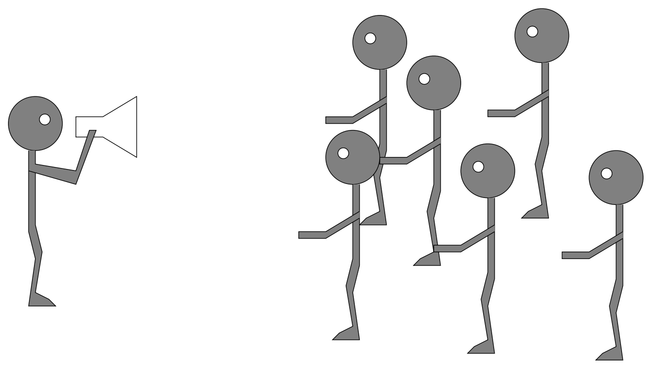

Figure 7: A broadcast sends a message to a number of recipients.

1.7.3 Reduction Operations

A reduction operation takes data items from several processors and reduces them to a single data item that is usually made available to all of the participating processors.

One example of a reduction operation is a strike vote where thousands of votes are reduced to a single decision. One common reduction operation in parallel programs is a summation over processors.

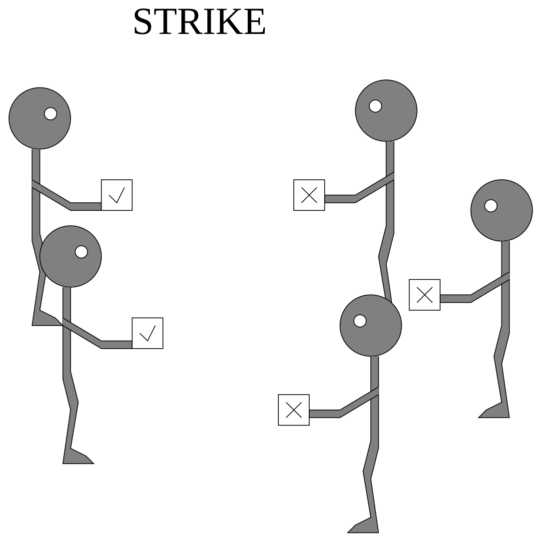

Figure 8: Reduction operations reduce data from a number of processors to a single item.

1.8 Introduction to MPI

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 workstations) 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).

1.9 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

2 MPI Programs

This section describes the basic structure of MPI programs.

2.1 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 (the EPCC implementation is described in "EPCC's MPI Implementation" on page 87).

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.

2.2 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.

2.3 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 77.

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.

2.4 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 [2], 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, handles are always of type `INTEGER` and arrays are indexed from `1`.

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 *`.

2.5 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
```

2.6 `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.

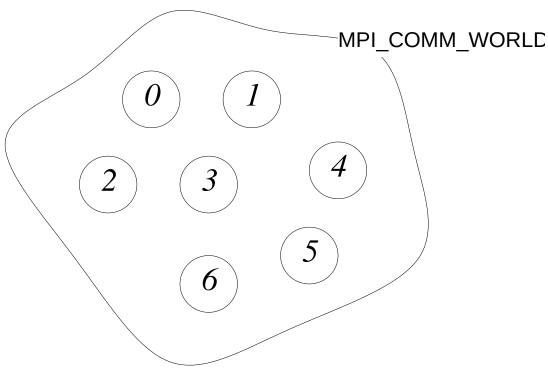

Figure 9: The predefined communicator `MPI_COMM_WORLD` *for seven processes. The numbers indicate the ranks of each process.*

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.

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 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.

2.7 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]

2.8 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.

2.9 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`[2]. Taking into account the previous two sections, it follows that *every* MPI program should have the following outline.

2.9.1 C version

```
#include <mpi.h>

/* Also include usual header files */

main(int argc, char **argv)

   {

      /* Initialise MPI */
```

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

2.In the EPCC implementation of MPI, the Fortran include file is called `mpif.inc`

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

    /* Main part of program .... */


    /* Terminate MPI */

    MPI_Finalize ();

    exit (0);

  }
```

2.9.2 Fortran version

```
      PROGRAM simple

      include 'mpif.h'

      integer errcode

C Initialise MPI

      call MPI_INIT (errcode)

C Main part of program ....

C Terminate MPI

      call MPI_FINALIZE (errcode)

      end
```

2.9.3 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)
```

2.10 Exercise: Hello World - the minimal MPI program

1. Write a minimal MPI program which print "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.
4. Modify your program so that the number of processes is printed out.

Extra exercise

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

3 What's in a Message?

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

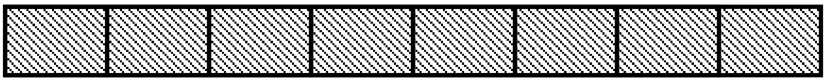

Figure 10: An MPI message.

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.

4 Point-to-Point Communication

4.1 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 communication in MPI, *collective* communication, which involves a whole group of processes at one time.

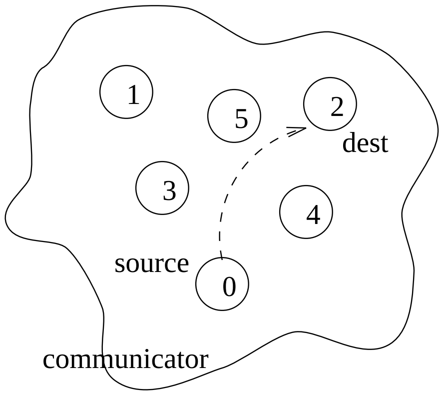

Figure 11: In point-to-point communication a process sends a message to another specific process

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.

4.2 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 80.

4.2.1 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.

4.2.2 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 12:. This acknowledgement must be received by the sender before the send is considered complete.

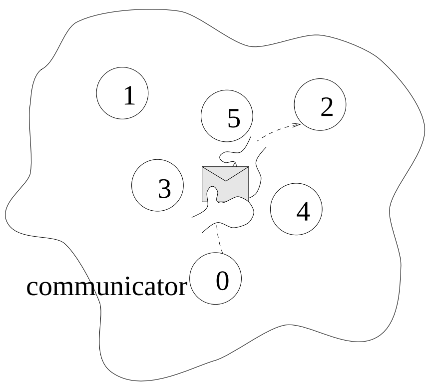

Figure 12: In the synchronous mode the sender knows that the other one has received the message.

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 26. Problems of unwanted synchronisation (such as deadlock) can be avoided by the use of non-blocking synchronous communication “Non-Blocking Communication” on page 35.

4.2.3 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 26. 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.

4.2.4 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 13:, 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.

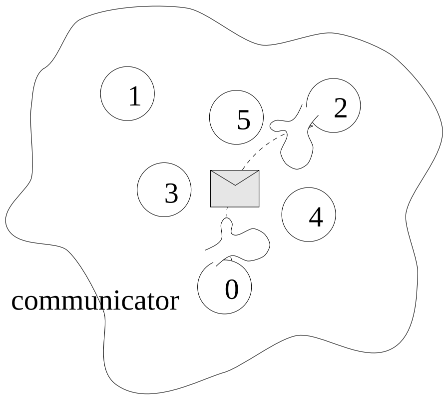

Figure 13: In the ready mode a process hopes that the other process has caught the message

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 14:

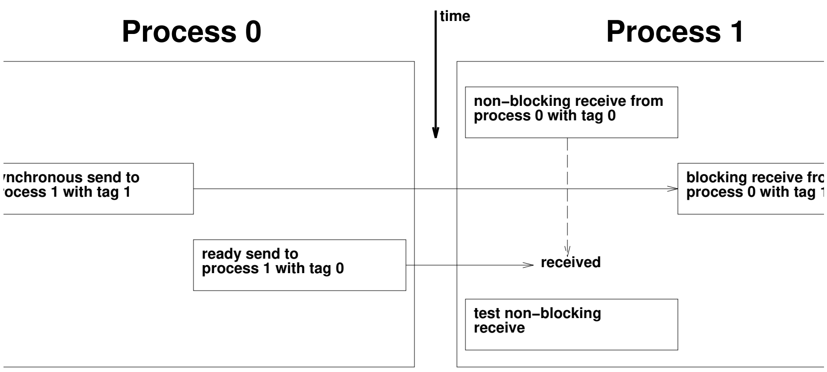

Figure 14: An example of safe use of ready mode. When Process 0 sends the message with tag 0 it ``knows'' that the receive has already been posted because of the synchronisation inherent in sending the message with tag 1.

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 35).

4.2.5 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.

4.3 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.

4.4 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`.

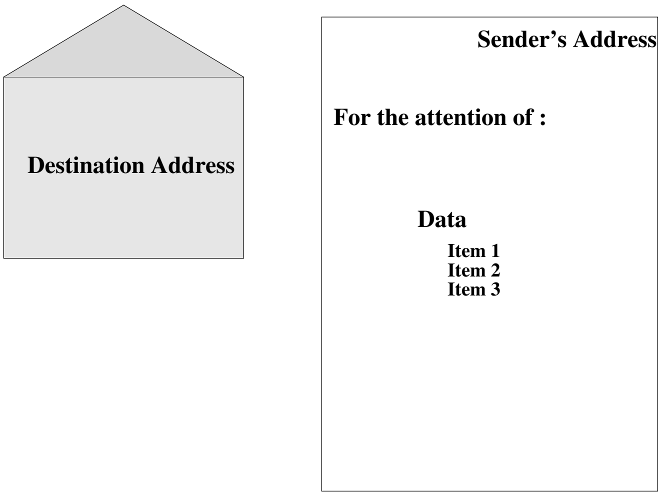

Figure 15: As well as the data, the message contains information about the communication in the communication envelope.

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.

4.4.1 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.

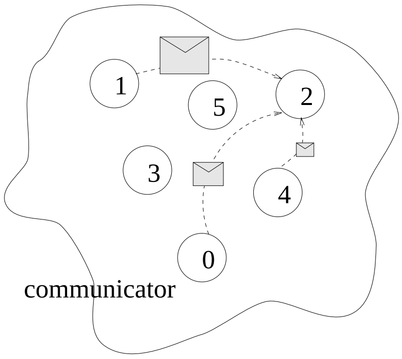

Figure 16: Processes can receive messages of different sizes.

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`.

4.5 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”).

4.5.1 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.

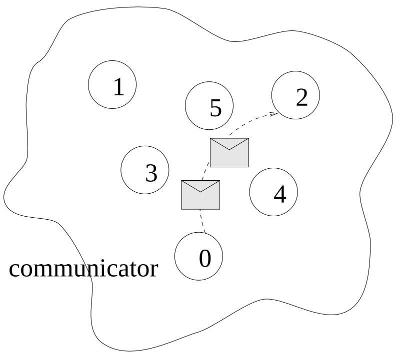

Figure 17: Messages sent from the same sender which match the same receive are received in the order they were sent.

4.5.2 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.

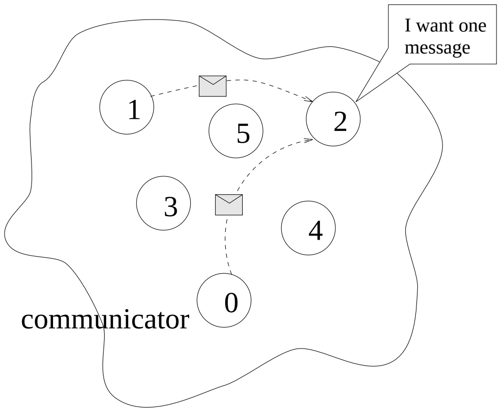

Figure 18: One communication will complete.

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.

4.6 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.)

4.7 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.

4.7.1 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.

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?

5 Non-Blocking Communication

5.1 Example: one-dimensional smoothing

Consider the example in Figure 19: (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.

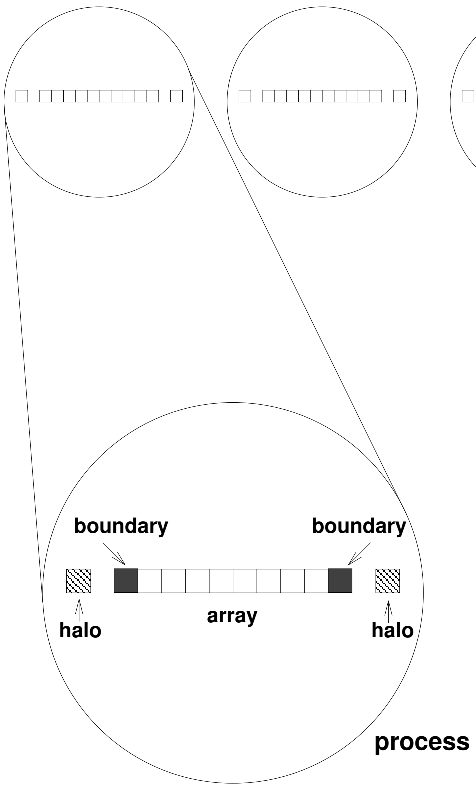

Figure 19: One-dimensional smoothing

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.

5.2 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.

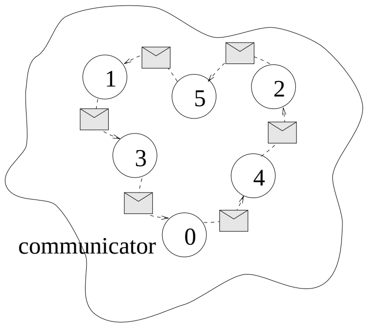

Figure 20: Deadlock

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 communication

1.Another solution might use `MPI_SEND_RECV`

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

(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].

5.3 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 80 describes an alternative way of expressing the same algorithm using persistent communications.

5.3.1 Non-blocking sends

The principle behind non-blocking *sends* is shown in Figure 21:.

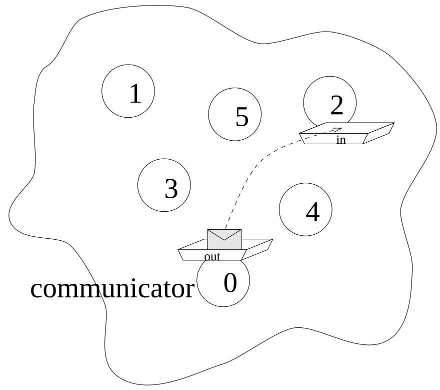

Figure 21: A non-blocking send

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 39.

5.3.2 Non-blocking receives

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

A non-blocking receive is shown in Figure 22:.

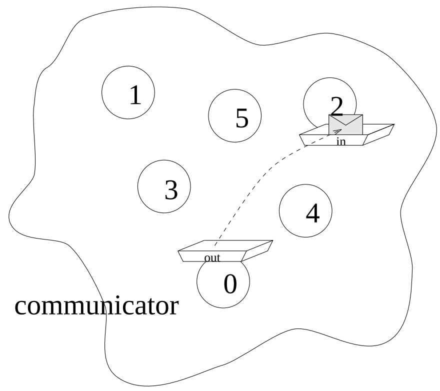

Figure 22: A non-blocking receive

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 39.

5.4 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.

5.4.1 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`.

5.4.2 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 23:). 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.

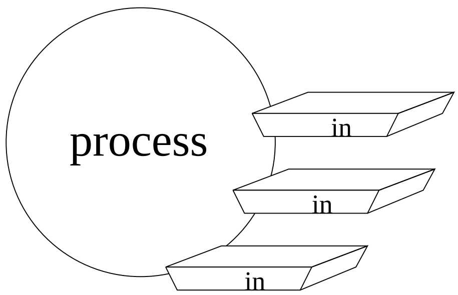

Figure 23: MPI allows a number of specified non-blocking communications to be tested in one go.

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.

5.4.3 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 24:).

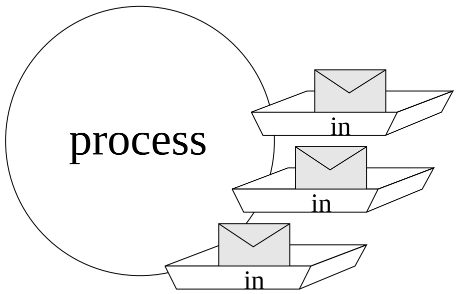

Figure 24: Test to see if all of the communications have completed.

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 31).

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.

5.4.4 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 25:).

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.

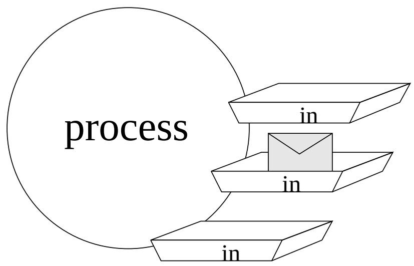

Figure 25: Test to see if any of the communications have completed.

5.4.5 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)
```

5.4.6 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.

5.5 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.

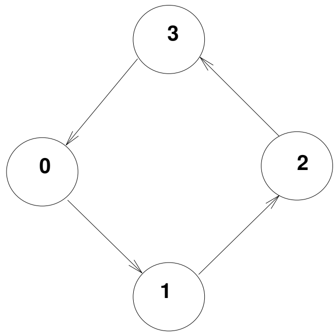

Figure 26: Four processors arranged in a ring.

Extra exercise

Modify the above program in order to measure 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 80— but do not deallocate in that case.

6 Introduction to Derived Datatypes

6.1 Motivation for derived datatypes

In “Datatype-matching rules” on page 33, 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.

6.1.1 Examples in C

6.1.1.1 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.

6.1.1.2 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.

6.1.1.3 A set of general variables

Consider

```
int nResults, n, m;

double results[RMAX];
```

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

6.1.2 Examples in Fortran

6.1.2.1 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.

6.1.2.2 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.

6.1.2.3 A set of general variable

Consider

```
INTEGER nResults, n, m

DOUBLE PRECISION results(RMAX)
```

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

6.1.3 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 43, 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.

6.2 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`.

6.2.1 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 [2].

6.2.1.1 MPI_TYPE_VECTOR

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

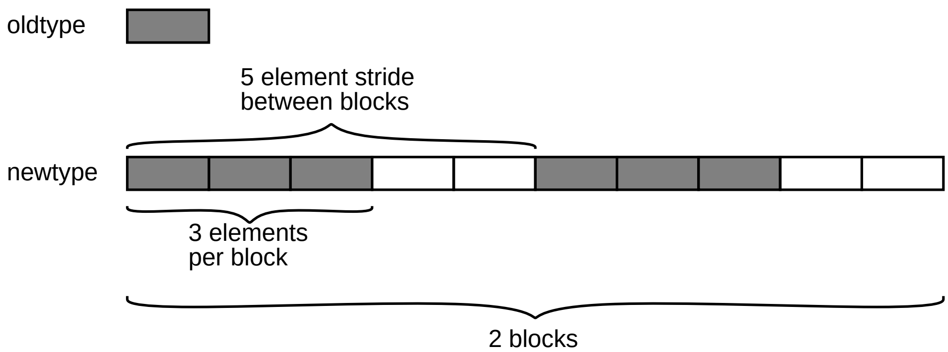

Figure 27: Illustration of a call to `MPI_TYPE_VECTOR` *with* `count = 2, stride = 5` *and* `blocklength = 3`

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 27:.

6.2.1.2 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 28:.

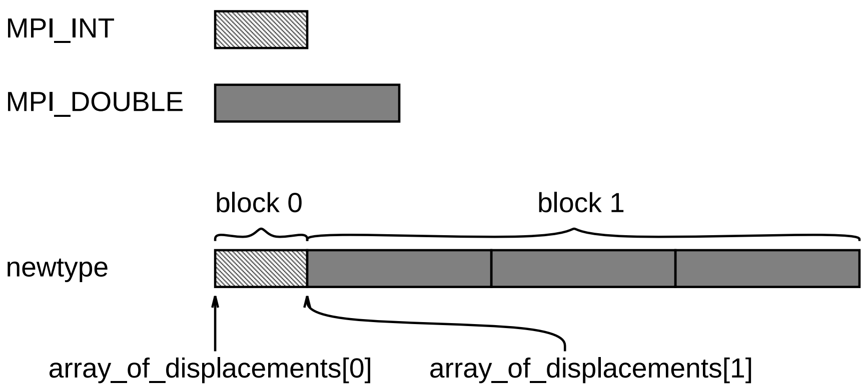

Figure 28: Illustration of a call to `MPI_TYPE_STRUCT` *with* `count = 2`, `array_of_blocklengths[0] = 1`, `array_of_types[0] = MPI_INT`, `array_of_blocklengths[1] = 3` *and* `array_of_types[1] = MPI_DOUBLE`

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

6.3 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`.

6.4 Example Use of Derived Datatypes in C

6.4.1 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][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);
```

6.4.2 A C `struct` (contiguous data of mixed type)

```
struct{
   int nResults;
   double results[RMAX];
} resultPacket;
/* ******************************************************** *
*     It is required 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 RESULT_PACKET_NBLOCKS 2
int array_of_blocklengths[RESULT_PACKET_NBLOCKS] = {1, RMAX};
MPI_Type_extent (MPI_INT, &extent);
MPI_Aint array_of_displacements[RESULT_PACKET_NBLOCKS] =
                                              {0, extent};
```

```
MPI_Datatype array_of_types[RESULT_PACKET_NBLOCKS] =
                                        {MPI_INT, MPI_DOUBLE};
/* ***********************************************************
* Use the description of the struct to construct a new
* type, and commit.
* ************************************************************ */
MPI_Datatype resultPacketType;
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);
```

6.4.3 A set of general variables (non-contiguous data of mixed type)

Unlike the contiguous case, the relative displacement between `nResults` and `results` cannot be known for certain. This case requires a different approach using the `MPI_ADDRESS` routine and the constant `MPI_BOTTOM`. These provide, respectively, the MPI address of a variable and the MPI address of the conceptual "origin" of the memory space. C programmers are recommended to use these rather than the native pointer arithmetic of C, for the sake of consistency with MPI.

```
int nResults;
double results[RMAX];
/* ************************************************************ *
* It is required to send nResults followed by results
* ************************************************************ */
```

```
int array_of_blocklengths[2] = {1, RMAX};

MPI_Aint array_of_displacements[2];

MPI_Datatype array_of_types[2] = {MPI_INT, MPI_DOUBLE};

MPI_Datatype newtype;

/* ********************************************************** *

* Find the addresses of nResults and results

* ********************************************************** */

MPI_Address (&nResults, &(array_of_displacements[0]));

MPI_Address (results, &(array_of_displacements[1]));

MPI_Type_struct (2,

                array_of_blocklengths,

                array_of_displacements,

                array_of_types,

                &newtype);

MPI_Type_commit (&newtype);

/* **********************************************************

* Addresses are relative to MPI_BOTTOM

* ********************************************************** */

MPI_Send (MPI_BOTTOM, 1, newtype, dest, tag, comm);
```

6.5 Example Use of Derived Datatypes in Fortran

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

```
     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 **************************************************

     INTEGER newtype

C **************************************************
```

```
C Construct a strided datatype and commit.
C JMAX blocks, each of length 1 element, separated by
C stride IMAX elements.
C oldtype = MPI_DOUBLE_PRECISION
C ***************************************************
      CALL MPI_TYPE_VECTOR (JMAX, 1, IMAX,
   &       MPI_DOUBLE_PRECISION, newtype, ierror)
      CALL MPI_TYPE_COMMIT (newtype, ierror)
C ***************************************************
C Use new type to send data, count = 1
C ***************************************************
      CALL MPI_SSEND (results(5, 1), 1, newtype, dest,
   &                           tag, comm, ierror)
```

6.5.2 A Fortran common block (contiguous data of mixed type)

```
      INTEGER RESULT_PACKET_NBLOCKS
      PARAMETER (RESULT_PACKET_NBLOCKS = 2)
      INTEGER nResults
      DOUBLE PRECISION results(RMAX)
      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

      array_of_displacements(1) = 0

      CALL MPI_TYPE_EXTENT(MPI_INTEGER, extent, ierror)

      array_of_displacements(2) = extent

      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 (2, 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)
```

6.5.3 A set of general variables (non-contiguous data of mixed type)

Unlike the contiguous case, the relative displacement between `nResults` and `results` cannot be know for certain. This case requires a different approach using the `MPI_ADDRESS` routine and the constant `MPI_BOTTOM`. These provide, respectively, the MPI address of a variable and the MPI address of the conceptual "origin" of the memory space.

```
      INTEGER array_of_blocklengths(2)

      INTEGER array_of_displacements(2)

      INTEGER array_of_types(2)
```

```
      INTEGER address(2)
      INTEGER newtype
      INTEGER nResults
      DOUBLE PRECISION results(RMAX)
C ***************************************************
C We want to send nResults followed by results.
C ***************************************************
      array_of_blocklengths(1) = 1
      array_of_blocklengths(2) = RMAX
C ***************************************************
C Find the addresses of nResults and results
C ***************************************************
      CALL MPI_ADDRESS(nResults, address(1))
      array_of_displacements(1) = address(1)
      CALL MPI_ADDRESS(results, address(2))
      array_of_displacements(2) = address(2)

      array_of_types(1) = MPI_INTEGER
      array_of_types(2) = MPI_DOUBLE_PRECISION
      CALL MPI_TYPE_STRUCT (2, array_of_blocklengths,
   &              array_of_displacements,
   &              array_of_types,
   &              newtype, ierror)
      CALL MPI_TYPE_COMMIT (newtype, ierror)
C ***************************************************
C Addresses are relative to MPI_BOTTOM
C ***************************************************
      CALL MPI_SSEND (MPI_BOTTOM, 1, newtype, dest, tag,
   &               comm, ierror)
```

Fortran programmers should note that there are conceptual issues surrounding the use of `MPI_BOTTOM` in Fortran. These are discussed in the "Unresolved Issues" section included at the end of the MPI standard [2]

6.6 Exercise

Modify the passing-around-a-ring exercise from “Exercise: Rotating information around a ring.” on page 42 so that it uses derived datatypes to pass round either a C structure or a Fortran common block which contains a floating point sum as well as the integer sum.

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?

7 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 29:), 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 optimisation, 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.

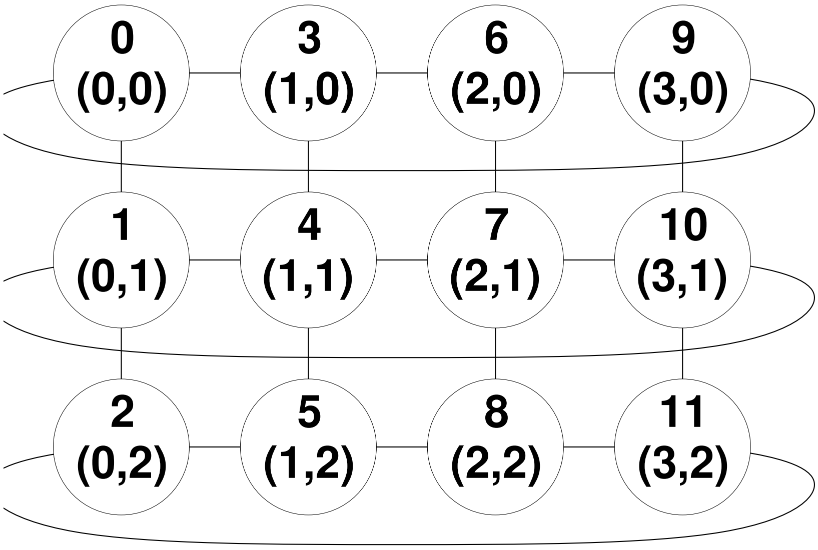

Figure 29: A virtual topology of twelve processes. The lines denote the main communication patterns, namely between neighbours. This grid actually has a cyclic boundary condition in one direction e.g. processes 0 and 9 are ``connected''. The numbers represent the ranks in the new communicator and the 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.

7.1 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 [2] for details.

7.2 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 77) 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 [2]).

7.2.1 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.

7.3 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 81). 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 30: and Figure 31:). Unfortunately, there is no provision for a diagonal "shift", although `MPI_CART_RANK` can be used instead.

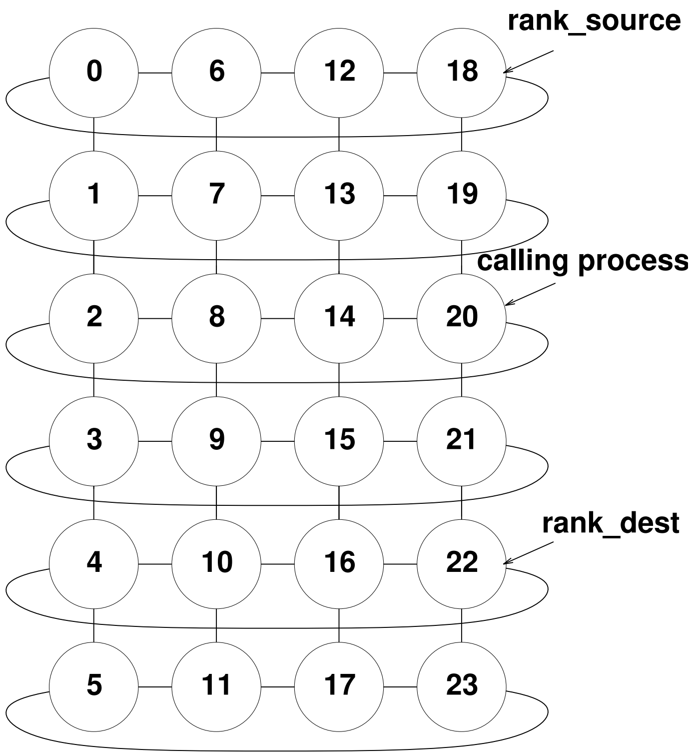

Figure 30: `MPI_CART_SHIFT` *is called on process 20 with a virtual topology as shown, with* `direction=0` *and with* `disp=2`

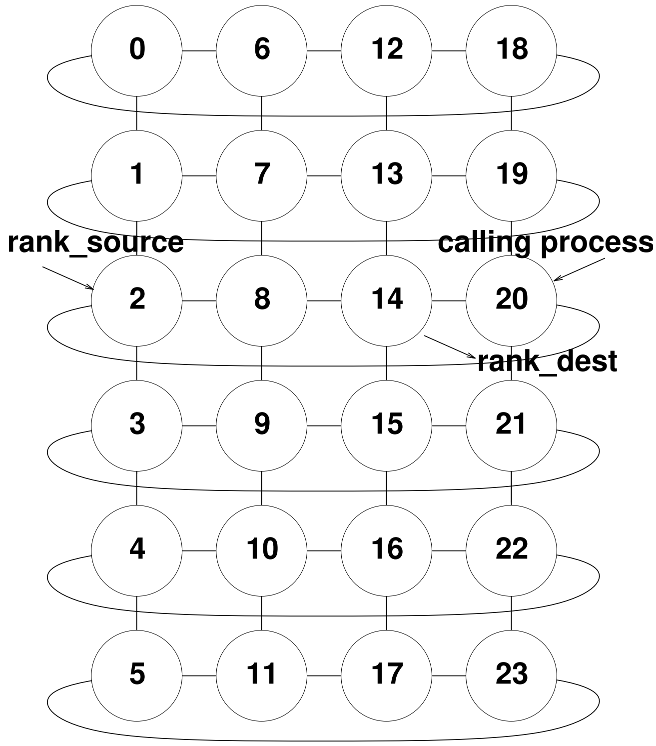

Figure 31: `MPI_CART_SHIFT` *is called on process 20 with a virtual topology as shown, with* `direction=1` *and with* `disp=-1`. *Note the effect of the periodic boundary condition*

7.4 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.

7.5 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.

7.6 Exercise

1. Re-write the exercise from page 54 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 right 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.

8 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 78). 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 23).
  - Datatypes must match between send and receive (see "Datatype-matching rules" on page 33).
- Differences include:
  - There is no concept of tags.
  - The sent message must fill the specified receive buffer.

8.1 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 performed 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.

8.2 Broadcast, scatter, gather, etc.

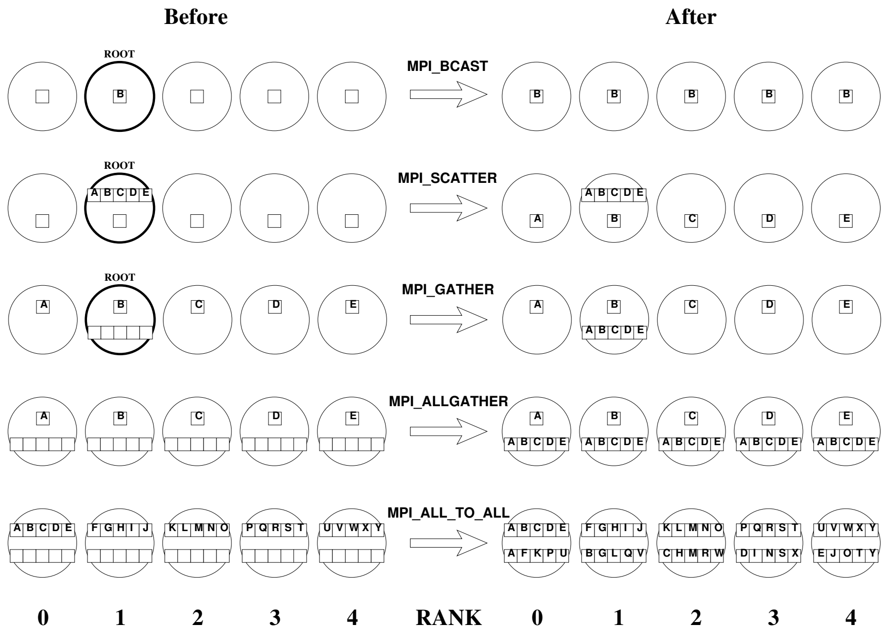

Figure 32: Schematic illustration of broadcast/scatter/gather operations. The circles represent processes with ranks as shown. The small boxes represent buffer space and the letters represent data items. Receive buffers are represented by the empty boxes on the ``before'' side, send buffers by the full boxes.

This set of routines distributes and re-distributes data without performing any operations on the data. The routines are shown schematically in Figure 32:. The full set of routines is as follows, classified here according to the form of the routine call.

8.2.1 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.

8.2.2 `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`).

8.2.3 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)
```

8.2.4 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 43) instead. Full details with examples and diagrams can be found in the MPI document [2].

8.3 Global reduction operations (global sums etc.)

8.3.1 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.

8.3.2 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 opera tions 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 `locations` "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 65).

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.

8.3.3 MPI_REDUCE

This is illustrated in Figure 33:.

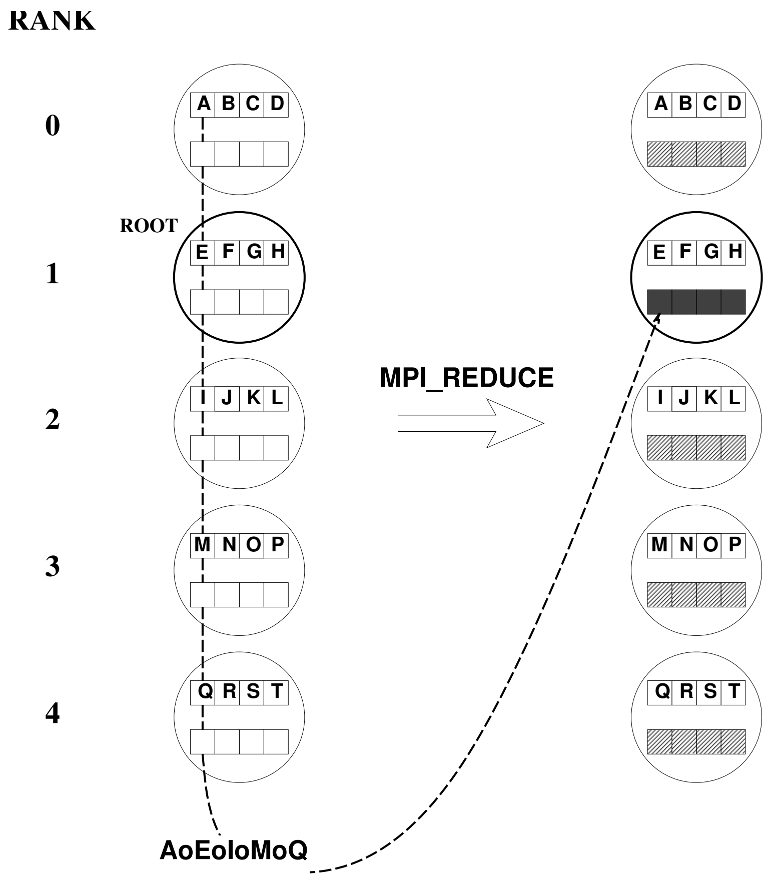

Figure 33: Global reduction in MPI with `MPI_REDUCE`*. o represents the reduction operator. The circles represent processes with ranks as shown. The small boxes represent buffer space and the letters represent data items. After the routine call, the light-shaded boxes represent buffer space with undefined 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 65 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.

8.3.4 Operators

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

8.3.4.1 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 63. More details with examples can be found in the MPI document [2].

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

8.3.4.2 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 a function of the following type

```
FUNCTION 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.

8.3.5 MPI_ALLREDUCE, MPI_REDUCE_SCATTER, MPI_SCAN

These are variants of MPI_REDUCE. They are illustrated in Figure 34:,

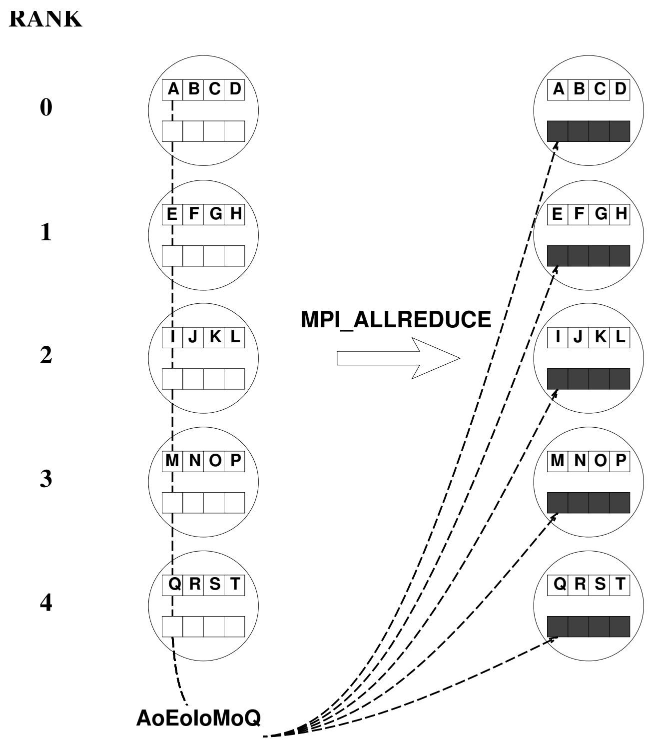

Figure 34: Global reduction in MPI with MPI_ALLREDUCE. The symbols are as in Figure 33:. The only difference from MPI_REDUCE is that there is no root --- all processes receive the result.

Figure 35:

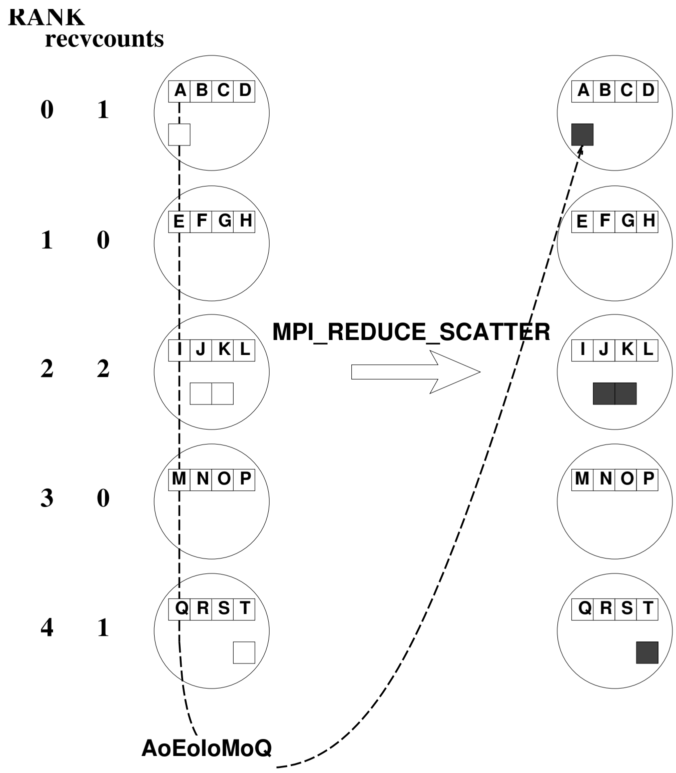

Figure 35: Global reduction in MPI with `MPI_REDUCE_SCATTER`*. The symbols are as in Figure 33:The difference from* `MPI_ALLREDUCE` *is that processes elect to receive a certain-size segment of the result. The segments are always distributed in rank order.*

and Figure 36:.

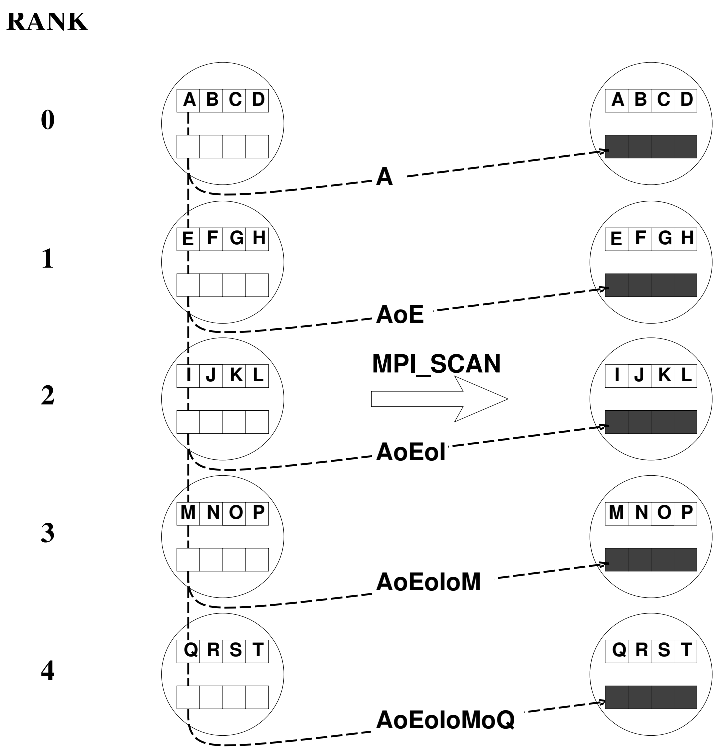

Figure 36: Global reduction in MPI with MPI_SCAN. The symbols are as in Figure 33:. The difference from MPI_ALLREDUCE is that the processes receive a partial result.

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 [2].

8.4 Exercise

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≥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?

9 MPI Case Study

9.1 A predator-prey simulation

The simulation of a predator prey model provides a good overview of the activities and challenges involved in parallelizing a realistic application code. Yet, the model is simple enough for the sequential code to be grasped quickly leaving you more time to practice with MPI.

9.1.1 What is the model?

Some foxes and rabbits live on a rectangular stretch of land. The problem is to find out how the two animal populations will evolve over the years. The simulation starts with some given initial population. Successive generations of foxes and rabbits are estimated over the years according to some simple predator-prey model. The problem is discretized by subdividing the whole land into square stretches of land of 1 kilometre of side. During a year, animals are born, some die, some are eaten and some other move in or out of a square stretch of land. Within each square stretch of land, an estimate of the population counts of foxes and rabbits is computed at the end of each year of the simulation.

Suppose there are *NS_Size* squares stretches of land along the North-South axis and *WE_Size* such stretches along the East-West axis. Let $f_{i,j}$ and $r_{i,j}$ denote respectively the number of foxes and rabbits in the i-j-th square patch of land at the beginning of the current year. The number of rabbits that survived until the end of the year is computed according to the formula shown below where birth, death and migration rates of rabbits (α_r β_r and μ_r respectively) are constant throughout the whole land and $\mu_r\Delta_r$ is a migration term. Similarly the corresponding estimate for foxes in the same square

$$\text{New } r_{i,j} = \alpha_r r_{i,j} + \beta_r f_{i,j} + \mu_r \Delta r$$

patch of land is given by the formula below where α_f, β_f and μ_f are constant throughout the whole land and $\mu_f\Delta_f$ is a migration factor.

$$\text{New } f_{i,j} = \alpha_f r_{i,j} + \beta_f f_{i,j} + \mu_f \Delta f$$

The boundary conditions are periodic in the space dimensions with period *NS_Size* in the North-South direction and with period *WE_Size* in the East-West direction. These periodic conditions are conductive of fox and rabbit populations that oscillate with time and are easily programmed.

9.1.2 What to do?

There are three steps to the case study: copy the tar file for the case study into your working directory, then compile and run the sequential program, called `ECO`, and parallelize the `ECO` program following the comments in either of the files `d2fox.c` or `d2fox.F` depending on which programming language you chose to use.

9.1.2.1 Getting the files

There are C and Fortran versions of the tar files for the MPI case study. They can be found in

```
/home/etg/MPIcourse/casestudy/C/casestudy.tar
```

and

```
/home/etg/MPIcourse/casestudy/F/casestudy.tar
```

Copy one of the two files into you working directory and then untar it as follows

```
tar xvf exercise.tar
```

9.1.2.2 The source code

The simulation program is called `ECO`. The version of the `ECO` program in the file whose name start with `n2fox` should run on any workstation. The `n2fox` file contains no explicit clue on how to produce an MPI version of the `ECO` program. Its purpose is to help you understand the data structures and the working of the sequential `ECO` program. The sequential ECO program can be compiled as follows:

```
make sequential
```

Depending on which programming language you chose, the file `d2fox.F` or `d2fox.c` is the recommended starting points for the case study. The `ECO` program in these files have been augmented with comments and some useful data structure have been declared. Some procedures in the files d2fox.c and d2fox.F have more arguments than their counterparts in the files n2fox.c and n2fox.F. These additions were made to let you concentrate on experimenting with MPI. Compile the baseline MPI ECO program with:

```
make
```

The program `ECO` in the file `ngfox.c` calls the `gnuplot` graphics package to plot the first five generations of foxes. This should give you some intuition about the predator-prey model. On some systems the data files for the first five generations of foxes will all be created first and then displayed one at a time. Viewing more generations of foxes may not be possible if your workstation disk space if nearly full. The corresponding executable can be compiled using the command:

```
make display
```

Whatever variant of the `ECO` program you compile, the makefiles provided with this distribution will generate an executable called *`fox`*.

9.2 The sequential ECO program

In accordance with the MPI usage, the name “procedure” designates either a C function or a Fortran subroutine. The number of foxes and rabbits in say the `i,j`-stretch of land (with `1≤i≤NS_Size` and `1≤j≤WE_Size`) are stored as the `i,j`-entries of the arrays `Fox` and `Rabbit` respectively. The initial population counts are computed by the procedure `SetLand`.

The procedure `Evolve` computes the populations of foxes and rabbits at time `k+1` from the populations at time `k`, for `1≤k≤NITER`. The two-dimensional arrays `TFox` and `TRabbit` store the new populations counts while these are being computed. Because of the periodic space boundary conditions, the new population counts for the first row of land stretches depend on old populations counts on first, second and last rows of land stretches. To avoid unnecessary conditional statements, both arrays `Fox` and `Rabbit` have a row indexed `0` that duplicates the last row of the corresponding array. Similarly both arrays `Fox` and `Rabbit` have a row indexed `NS_Size+1` and columns indexed `0` and `WE_Size+1`. The data in the additional rows and columns is called *halo data*. The procedure `FillBorder` is called at the beginning of procedure `Evolve` for each animal population to handle halo data. Most communication in the parallel ECO program take place within the procedure `FillBorder`.

The total population counts of foxes and rabbits are computed once every `PERIOD` years by calling twice the procedure `GetPopulation`, once for each animal population.

9.3 Toward a parallel ECO program

9.3.1 A 1-D cartesian topology.

1. Run the sequential `ECO` program in either of the file `d2fox.c` or `d2fox.F` on a single processor. Save the output of the simulation.

2. Define a 2-D cartesian topology in procedure `SetMesh`. Mimic a 1-D topology by setting the extent of one of the dimensions of the process mesh to `1`. Preferably, choose that dimension that will result in messages of unit-stride, or contiguous, vector data type being exchanged between processes.

3. Define a geometric data decomposition in procedure `SetLand`.The variables *fx* and *lx* record respectively the global indices of the first and last rows of the arrays Fox and Rabbits for the current process. The variables *fy* and *ly* store the corresponding global indices of columns for these two arrays. These four variables are related by the two equations shown below where the variables *dx* and *dy* are set appropriately in procedure `SetLand`. The values of all four variables

$$lx = fx + dx$$

$$ly = fy + dy$$

fx, *lx*, *fy* and *ly* are saved into the array *lbnds* at the end of procedure `SetLand` for later use by other procedures.

4. Define in procedure `SetComm`, the MPI derived datatypes for shifting halo data along the ring of processes. The MPI derived datatype for shifting a column is stored in the entry *COLUMN* of the array named `types`. The procedure `SetComm` is also responsible for storing in the array *neigh* the precomputed ranks of

nearest neighbour processes.

5. Write in procedure `FillBorder`, the necessary MPI send and receive procedure calls to shift halo data between nearest-neighbour processes.

6. Complete the procedure `GetPopulation` to sum the local population counts computed by each process.

7. Check that your parallel simulation produces the same results as the sequential one.

9.3.2 On a 2-D process grid.

Make your code work with an arbitrary the 2-D cartesian topology. You might find it useful in a first instance to set to 1 the extent of one of the two dimensions of the process mesh before letting MPI_Dims_create determine the extend of both dimensions.

9.4 Extra exercises

These additional exercises are fairly independent of one another, and are roughly ordered in increasing amount of work.

1. Rewrite the data shifts in procedure `FillBorder` using the `MPI_Sendrecv` procedure.

2. Make the size of the land small, e.g. `NS_Size=WE_Size=4`, and run your simulation with various number of processes. Do you get the same result on every run? If not, can you fix the problem by adding appropriate `else`-conditions in procedure `FillBorder`?

3. Time your program and see how the shape of the 2-D process mesh affects the run time of the simulation. You might get more interesting results if you do not time the print-statement in the main loop of the main program.

4. Replace the data shifts in procedure FillBorder by persistent communication requests. You could, for example, call the procedures `MPI_Send_init` and `MPI_Recv_init` from in procedure `SetComm`. Does your new parallel ECO program run faster than your previous parallel ECO program? Does your conclusion still hold when you increase either the number of processes or the size of the whole land?

5. Overlap communication and arithmetic. Compute the new local population counts that do not depend on halo data while halo data is being sent and received. What happens when you increase either the number of processes or the size of the whole land?

6. Simulate the fox and rabbit populations on a sphere. Model each animal population on the southern and northern hemispheres by two separate 2-D arrays. Assign the equator to the northern hemisphere. Do not change the equations, so that your current 2-D parallel ECO program can readily be adapted to compute fox and rabbit populations in the interior of each hemisphere.

10 Further topics in MPI

10.1 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[2] (see `MPI_ERRHANDLER_CREATE`, `MPI_ERRHANDLER_SET`, `MPI_ERRHANDLER_FREE`) but are not discussed in this course.

10.2 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.

10.3 Communicators, groups and contexts

10.3.1 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 37:). Contexts define the scope for communication.

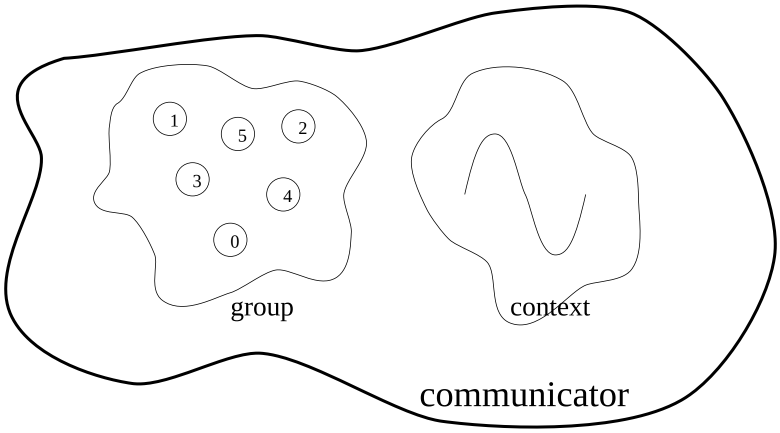

Figure 37: A communicator.

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.

10.3.2 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 (see "Collective Communication" on page 61) 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.

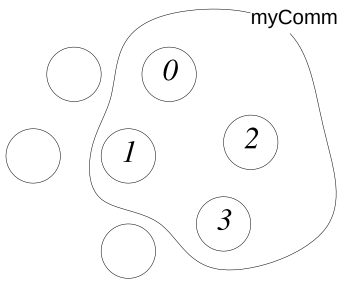

Figure 38: A new communicator defined on a subset of the processes in `MPI_COMM_WORLD`.

There are other reasons for creating a new communicator. When creating a virtual topology (see "Convenient Process Naming: Virtual Topologies" on page 55), 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.

10.3.3 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.

10.3.4 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 61 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 61) 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.

10.3.5 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.

10.4 Advanced topics on point-to-point communication

10.4.1 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 31.

```
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.

10.4.2 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 39. The only difference to the use of the non-blocking communication routines in“Non-Blocking Communication” on page 35 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 35 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_SRART;

   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.

10.4.3 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 Figure 39: 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 36 for a quick description of "red-black").

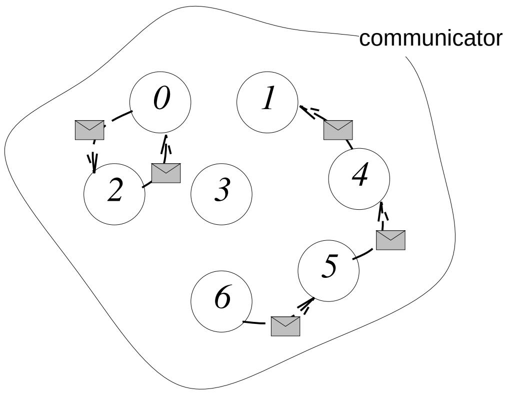

Figure 39: An example of two shifts. `MPI_SENDRECV` *could be used for both*

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 39: 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 39: could work thus with `MPI_SENDRECV`:

Table 8: Communications from Figure 39:

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

11 For further information on MPI

Two books are in preparation at the time of writing, namely [3]. A newsgroup `comp.parallel.mpi` has been set up. World Wide Web access to the MPI FAQ ("frequently asked questions") is via

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

and the MPI document [2] itself is available through WWW via

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

12 References

[1] R. Alasdair A. Bruce, James G. Mills and A. Gordon Smith, ''CHIMP/MPI User Guide''. Edinburgh Parallel Computing Centre, 1994. EPCC-KTP-CHIMP-V2-USER.

[2] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, 1994.

[3] William Gropp, Ewing Lusk and Anthony Skjellum. ''Using MPI: Portable Parallel Programming with the Message Passing", MIT Press, 1994.

Appendix A: EPCC's MPI Implementation

A.1 How to compile and run MPI code

The user guide for the EPCC implementation of MPI is [1]. (The EPCC implementa tion of MPI used in this course uses a lower level of software known as "CHIMP" and references to "CHIMP" may therefore appear in error messages etc.)

A.2 Modifying your environment

The mpiinst script will set up your environment for MPI. *It is only necessary to this the first time you use MPI on your system.* For example, assume the name of the directory in which MPI is installed is mympi. Then you should type

```
mympi/bin/mpiinst
```

You will find that the mpiinst has appended lines to your startup files (.login, .profile or whatever) and created a new symbolic link called .chimpv2rc. You must now either source your startup file, or log out and then log in again.

A.3 Compiling MPI Code

The EPCC version of MPI provides scripts for the compilation of MPI code, one for C, mpicc, and one for Fortran, mpif77.

When the compilation script is run, it compiles the program for the architecture it was called on. If you wish to run the code on a different architecture this must be specified with the -mpiarch option.

To compile the MPI program simple.c for a Sun workstation, the compile line would be:

```
mpicc –mpiarch sun4 –o simple simple.c
```

For the equivalent Fortran code, simple.F, the compile line would be:

```
mpif77 –mpiarch sun4 –o simple simple.F
```

Details of other options for the compilation scripts can be found in [1].

A.3.1 Configuration Files

In order to run an MPI program using the EPCC implementation of MPI you require a configuration file. This file specifies which executables (processes) the loader should load onto which processors ("hosts"). The name of the configuration file is usually of the form <name>.cfg. A configuration file, simple.cfg for the example programs shown in “A simple MPI program” on page 13 could be as follows:

```
# The configuration file for one instance of the program
# ‘simple’.
(simple) : type=sun4
```

This will execute the executable simple on the processor on which the MPI loader is called (if the host is not of the correct type this is likely to fail!). The structure of the configuration file is as follows:

- The name of the executable for the MPI process is enclosed in brackets.
- Command-line arguments can be given to the executable by including them in side the brackets after the executable name in the normal way.
- The architecture you wish to run the code on is specified to MPI in the form type=<architecture>. For example, if you wish to run the code on Sun workstations, you would specify the architecture as type=sun4. The type is used by MPI to set its parameters and configure itself to support message-pass ing on each particular host architecture.
- The type= syntax is an example of a configuration file attribute and is there fore separated from the list of processes by a colon.
- Comments can be included in a configuration file by preceding them with #.

More details on how to construct configuration files can be found in [1].

A.3.2 Loader Command

The loader reads the configuration file and loads the processes onto the specified resources. The loader command has the following form:

```
mpirun <options> <configuration file>
```

Details of the possible options are contained in [1]. One particularly useful option is the verbose option -d which provides information on what the loader is doing.

To run the configuration file simple.cfg described in the previous section the loader command would be:

```
mpirun simple.cfg
```

or with the verbose option:

```
mpirun -d simple.cfg
```

A.3.3 Running Multiple Instances of the Same MPI Process

It is common to run several processes all consisting of the same executable. For exam ple, if we wish to run three instances of the process work the configuration file, work.cfg, could include the following line:

```
3 (work) : type=sun4
```

While this does what is required, it may be the case that we want to run different numbers of work on different occasions. Rather than having to edit the configuration file every time a different number of processes are required, it is simpler to pass the number of instances required as an argument to the loader. This is done as follows:

```
$1 (work) : type=sun4
```

In this case e.g., for three instances of work the command line would be

```
mpirun work.cfg 3
```

Because no host attribute has been specified, the set of processors onto which these processes will be loaded depends on details of "resource management". For example, for a Unix workstation environment, all processes will be loaded on the "local host" i.e. the workstation on which mpirun was called, but on a specialised parallel machine the processes might be loaded onto different processors. Resource manage ment is not discussed in this course. For the case of Unix workstations, of course the processes have to be loaded onto different hosts to make a truly parallel program. For example, to load a copy of the executable work onto each of the local host and two remote hosts (blair and coventry) the following configuration file could be used:

```
work : type=sun4
work : type=sun4, host=blair
work : type=sun4, host=coventry
```

Appendix B: Machine Specific Information

The machine used in this course in a T800 Meiko Computing Surface, ssp.epcc.ed.ac.uk, which is housed here in EPCC. To login to it from the EPCC training room workstations, type

```
rlogin ssp
```

The machine contains around 150 T800 processors which are arranged into domains which range in size from 2 processors to 150 processors. In order to run code on the ssp, a user must be 'attached' to a domain. Only one user can attach to a domain at any one time, so you have sole access to the processors in a domain during the time you are attached to it. Information on the domains, and whether they are available can be obtained using the csusers command.

```
user@ssp$ csusers -a

VRM version 1.16c; last loaded: Sun Jan 22 13:07:20 1995


Resource        User            Attached
d2a             AVAILABLE
d2b             AVAILABLE
d2c             AVAILABLE
dm8             AVAILABLE
d17a            AVAILABLE
d17b            AVAILABLE
d17c            AVAILABLE
d17d            AVAILABLE
d17e            AVAILABLE
d17f            AVAILABLE
d17g            AVAILABLE
dm17            AVAILABLE

Class           Members
d68             d68a d68b
d51             d51a d51b d51c d51d d51e d51f
d34             d34a d34b d34c d34d d34e d34f d34g d34g
d17             d17a d17b d17c d17d d17e d17f d17g
d2              d2a d2b d2c

Additive Domain Members
d150            d2a d2b d2c dm8 d136
d136            d17a d17b d17c d17d d17e d17f d17g dm17
d68b            d17e d17f d17g dm17
d68a            d17a d17b d17c d17d
d51b            d17e d17f d17g
d51c            d17b d17c d17d
d51d            d17a d17b d17d
```

```
d51a            d17a d17b d17c
d34g            d17f d17g
d34e            d17e d17f
d34d            d17b d17c
d34c            d17a d17d
d34b            d17c d17d
d34a            d17a d17b
```

The domain names have the form

```
d<number><letter>
```

The letter d identifies it as a domain on the ssp, while the number is the number of processors in the domain, and the letter differentiates between a number of different domains with the same number of processors in them. The larger domains are made up of combinations of the smaller domain.

B.1 Requesting Resources

A user connects to a domain using the csattach command followed by the name of the domain the user wishes to attach to. For example, if you wish to attach to a domain with name d5a then the command would be

```
csattach d5a
```

Alternatively, if the user is not concerned which of the group domains they are attached to, they can csattach to the generic domain. For example in the above example the generic domain would be d5 and the user could attach to it using the following command

```
csattach d5
```

which will result in the user be given any of that class of domains which is free and to which they are permitted access.

Once a domain has been attached to sucessfully, the following message is printed

```
user@ssp$ csattach d17
Request for d17 granted.
d17a: attaching to 17 x T800
Total remaining allocation: 3294:12:21 processor hours
Timeout on this connection limited to: 193:46:36 hours
user@ssp$
```

The user has exclusive access to the processors in the domain, in this case d17a, until they either detach from the domain, or logout of the machine. A user can detach from a domain using the csdetach command.

```
user@ssp$ csdetach
d17: detached
Connect time = 0:01:15; processor time = 0:21:15
Total remaining allocation: 3293:51:06 processor hours
user@ssp$
```

As well as freeing up the resources for other users, csdetach gives information about the amount of time you were connected to the domain, and how much processor time was used.