THE UNIVERSITY
of MANCHESTER

Introduction to MPI

Student Notes

Martin Preston

Manchester and North HPC T&EC

MAN T&EC
The Manchester and North HPC Training and Education Centre

Edition 1.0 January 1996

1 Introduction

This document contains the student notes for the MAN T&EC course “Introduction to MPI”. These notes are intended to supplement the course, rather than provide a sufficient introduction to the topic on their own.

The course covers:

- The basic concepts of message passing libraries, and how a library must provide more than basic data transmission primitives.
- The background to, and history of, the development of the Message Passing Interface (MPI).
- The basic concepts of MPI, and how these affects how applications can be constructed.
- How current (July ‘94) implementations may be used.
- The MPI message, and how these can be constructed and passed between executing programs using MPI calls.
- An introduction to the derived datatype features of MPI.

The Appendices cover:

- The example exercises used during the course, with full solutions and discussions of how they work.
- A resource list pointing the way to more comprehensive MPI material.

2 Message Passing

Message Passing is simply one of many parallel programming paradigms which are currently popular. Though it is often seen as specific to use on MIMD machines in reality it is equally suitable as a way of programming networked connections of workstations as it is high power, high expense, MIMD supercomputers.

A message passing application is composed of a number of component programs. Each program operates (and often executes) completely independently, perhaps on different processors. Message passing gets its name from the way in which each independent program communicates with the outside world, during its execution, by exchanging messages.

In this course we will concentrate on describing one particular message passing library, MPI, which provides pre-written functions to make this message transmission process relatively easy. Does this mean that all you need to know in order to write message passing programs is the function names of a particular library?

Unfortunately no! Before we can discuss the actual message transmission process we must first outline some of the ramifications of splitting our application into multiple programs.

2.1 Application Logic

All computer programmers will be familiar with the idea that his or her application has a particular application logic, some of which is explicitly controlled by the programmer, and some of which is implicitly expressed (perhaps as a virtue of the application being written in a certain language).

For example if a C programmer writes the following lines of code he or she is encoding some logic into the application.

```
if(index <= 10)
   index = index + 1;
else
{
   index = 0;
   printf("Finished\n");
}
```

In this case the programmer is specifying

- If index is below the specified threshold then increment it.
- If not *first* reset it *then* print a message.

Much of this will seem intuitively obvious to the experienced programmer, especially the concept that there is an ordering to the statement execution (do something then do something else). However if we split this simple application into a number of programs much of this implicit application logic is lost.

```
for(time=start; time++; time<end)
{
   process_all_bodies();
   display_bodies();
}
```

Figure 1: Serial application

```
for(time=start; time++; time<end)
   process_all_bodies();
```

First Program

```
for(time=start; time++; time<end)
   display_bodies();
```

Second Program

Figure 2: Initial application decomposition

For example we may have written a computer animation program, and somewhere within the code we may have some lines which look like the program shown in Figure 1.

We may decide that this loop takes a great deal of computation, and may choose to parallelise it by using message passing.

The programmer may decide that each of the function calls takes a large amount of time, so it may be wise to split each function call into a separate program. In this case the resulting programs might look like those shown in Figure 2. This initially seems fine, and might initially seem plausible.

Notice, however, that our application relies on the implicit assumption that at each iteration we must process the bodies THEN display them. If we split the application in the manner shown in Figure 2 then we lose this guarantee, each program can operate completely independently. Clearly we do not want this!

For message passing, i.e., a system where we split our application into independent programs, to be useful we require the ability to control the *application logic*, in this case the timing of the two programs.

In our animation example we need to need to decompose our application in the way shown in Figure 3, in other words at each iteration the second program must wait for the first program to finish its line of the master loop before it can start.

```
for(time=start, time++; time<end)
{
   process_all_bodies();
   tell program 2 to display;
}
```

```
for(time=start; time++; time<end)
{
   wait for program 1;
   display_bodies();
}
```

Figure 3: Sensible Decomposition

2.2 Facilities of MP

We have seen, then, that message passing applications require more than the ability just to send data, or messages. In practice all message passing libraries (of which there are a large number) provide a key set of facilities for the application developer.

These include:

- The ability to create processes on remote machines, i.e., the way in which we would cause the two programs in our previous example to start executing on separate processors.
- The ability to monitor the state of those processes. For example we might want to know whether a remote process is still executing, and finally,
- Routines which enable programs to send messages, or signals, to other programs.

All message passing libraries provide these facilities, though the terminology will vary between libraries.

2.3 Message Passing Libraries

Message Passing, as a programming paradigm, has been around for a long time, and over the years has attracted a large amount of interest from the parallel academic community. Unfortunately in many respects message passing has suffered from this degree of interest!

As previously mentioned the naive view of message passing is that of transmitting data between processes. As fundamentally the development of libraries which can do this is so easy (as most modern OS's & networks already provide low level interfaces to these sort of facilities) a large number of message passing libraries have been developed.

Each of which was developed for a particular application (which the developers were working on at the time), and was then made available to the academic community at large as a research resource. Whilst this release of the products of research is normally a good thing it becomes a hindrance when large numbers of message passing libraries become available, few of which have been developed adequately!

Until recently, then, the parallel developer coming to message passing for the first time had to make a choice between a large number of implementations, all of which performed the same job, but were mutually incompatible!

In recognition of this the parallel community decided to resolve the issue by developing a message passing standard, which would specify the interface a programmer has to the library. If a developer chooses to use a library which conforms to this standard then he or she should be able to move to a different implementation of message passing (which conformed to the same standard) with only a recompilation.

The development of a standard interface, rather than a standard library (i.e., a piece of C or Fortran code which can be compiled on a wide range of machines) has two big advantages:

- If the standard only specifies the interface, then future research work can be performed on improving implementations without breaking the spec. (i.e., MP libraries will get faster without the application needing to be changed).
- Parallel computing vendors can employ implementations which suit their particular architecture, but applications can still be moved between architectures easily. So, for example, a particular vendor who uses large data buffers can use an implementation which sends large messages efficiently, while another ven-

dor can use a different strategy.

Having agreed that this was a sensible approach to the advancement of message passing the development process began,

2.4 The Message Passing Interface (MPI)

The MPI committee began meeting in April 1992, and met every six weeks during '93. The group consisted of over 40 Universities, various commercial vendors (of both hardware and software) and key message passing users (such as research institutes).

Rather than attempting to develop a wholly new way of dealing with message passing libraries the group decided to adopt the best features of a number of MP implementations, notably:

- Intels' NX/2
- Express
- PVM
- p4
- CHIMP (From Edinburgh Parallel Computing Centre)
- Work at IBM TJ Watson Research Centre
- PICL

Note that none of these standards was adopted as the basis, they were simply used as a source of ideas for features. Whilst this means that code written for any of the existing libraries will not transfer to MPI instantly, it means that any programmer experienced with these will find things in MPI that seem familiar!

The finished MPI standard which, remember, is a piece of paper, not software, contained descriptions of the interfaces for:

- Point to Point communication.
- Collective communication routines.
- Support for grouping operations, i.e., ways of telling an MPI-compliant library about the way in which your programs are organised.
- Mechanisms for separating communications within applications.
- Bindings for C & Fortran77
- A profiling interface.

2.5 What ISN'T in MPI

In an attempt to produce a description of MPI reasonably quickly the committee decided to deliberately leave some functionality out of the first MPI standard. These include:

- Explicit shared memory operations, i.e., ways in which your application can take advantage of operations of specific architectures.
- Any support for process management, and we will talk about this later!
- Support for threads.
- Debugging facilities.
- Parallel I/O.

At the time of writing the MPI group are preparing MPI2, which hopefully should address these issues. At the moment MPI doesn't address all the things that every MP user will want.

2.6 Implementations of MPI

MPI, as we have mentioned, is simply a standards document, which specifies the *interface* to a particular implementation. When we wish to develop applications using MPI we must actually work with one of these implementations.

At the time of writing almost all MPI applications are written using public domain MPI libraries. This is partly because MPI hasn't been around for very long, but is also a product of the MPI development process.

During the MPI standardisation a large number of ways of achieving particular functionality was discussed. In order to evaluate the relative merits of competing proposals a number of the academic members of the group developed proof-of-concept implementations.

To make this process easier these implementations were actually wrappers which sat on top of existing message passing libraries. So an application might call an MPI procedure, but internally to the library this resulting in a call to another library to actually perform the work. This has advantages and disadvantages!

The advantage is that, when the MPI standard was finished, there were already implementations which developers could begin using, presumably while waiting for the optimised vendor-specific implementations to become available. The second main advantage, and the one which is of most interest to us, is the ability they have to plug the gaps in MPI.

As we have seen the MPI standard doesn't address some important issues, notably process management. However any MPI implementation must provide some way of getting at this functionality (otherwise the implementation would be useless!). Therefore the PD implementations achieve this by simply letting the application call the lower level functions in the base MP library, so an application would spawn new processes by simply calling a non-MPI function (Figure 4).

This isn't ideal but is bearable for the time being.

The principal disadvantage is that of all PD software, largely poor quality documentation and limited support!

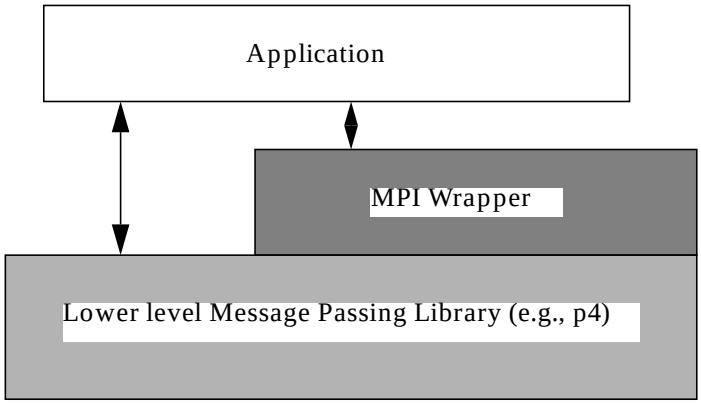

Figure 4: PD MPI Schematic

2.7 Summary

It is worth remembering that MPI is merely a standards document, when you are producing an application you are working with an implementation, *not MPI* itself.

Having described the background of message passing, some of the issues of constructing and using message passing, and the development process of MPI it is possible to look at how an MPI compliant library is used by the programmer.

3 Fundamentals of MPI

In this chapter we will discuss the basics of constructing an MPI application, including the terminology used within MPI.

3.1 The SPMD Model

As we have seen the MPI spec. doesn't specify how an application can cause other processes to begin executing in a standard manner. Obviously we can use the facilities of the underlying message library to achieve this, but this would mean our application would become unportable.Fortunately we can construct our application in a way that enables us to avoid this problem, using the Single Program, Multiple Data (SPMD) model.

The problem with the current MPI spec is that is difficult to start executing *different* programs, however all MPI implementations allow the user to cause the execution of several copies of the *same* program.

Therefore if we wish to construct portable MPI programs its advisable to write *one* program, which decides at run time what the program is actually going to do. In practice this isn't the major problem it initially appears. Instead of structuring our application as separate programs we write one piece of source code, which on start up decides which function to call.

For example rather than write our earlier animation example as two programs (as shown in Figure 3) we might write two functions, which are chosen depending on which processor this particular program is executing:

```
if(executing on root processor)
   animation_process();
else
   graphics_process();
```

In larger applications the process may be somewhat more complicated, but fundamentally we are making a decision dependent on *where* the program is executing.

3.2 MPI Preliminaries

A users application makes use of an MPI implementation by calling MPI functions. As previously mentioned the MPI standard defines bindings for C & Fortran77, and in practice the two bindings are extremely similar.

- Functions in both bindings are prefixed with the letters `MPI_` to help programmers find MPI calls in large programs.
- The C versions of the function names are in mixed case, whereas all the Fortran function names are in UPPER CASE.
- Each of the MPI functions returns an error, or information, code. In the C bindings these are returned from the function, whereas in Fortran an extra parameter

is passed to the function, and upon completion this variable will be set to the relevant value.

Whilst MPI implementations try to hide as much of their internal workings as possible there are a few occasions where it is useful to pass some data to the application which is only meaningful to a particular MPI implementation.

For example in MPI, as we will see, it is possible to start an operation, do some useful work in the application, and then have the application ask the MPI library whether the operation was successful. In order to do this the MPI library passes a token back to the application which refers to that operation. These tokens are called *handles* by MPI.

In Fortran these handles are simply integers, whereas in C each particular type of handle is `typedef`'d separately.

Finally in Fortran the arrays we pass to MPI are assumed to begin indexing from 1, whereas in C they start at entry 0.

3.3 Communicators

As we have seen message passing libraries provide a range of facilities, but possibly the most important is the manner in which they allow you to send data between programs. It is therefore necessary to see how MPI deals with message passing issues.

In MPI messages can only travel within a *communicator (*Figure 5*)*. These communicators are simply groups of processes which are assumed to know about one another. When a program decides to send a message to another process it does this by calling an MPI function. All MPI messaging functions require the programmer to specify a certain number of pieces of information:

- Firstly the data we will send (more on this later).
- The communicator within which the message will pass, and
- The *rank* of the destination process.

Within a communicator a process has a rank, which is essentially just a number which is unique to it *within this communicator.*

Why the restriction on having a rank which is unique to a communicator? In MPI a process can be a member of more than one communicator, and so in order to address a particular process as a destination for a message it is necessary to specify both the communicator, and the rank of the destination *within that communicator.*

In MPI an initial communicator is always created which is called `MPI_COMM_WORLD`. However MPI provides functions which enable new communicators to be created.

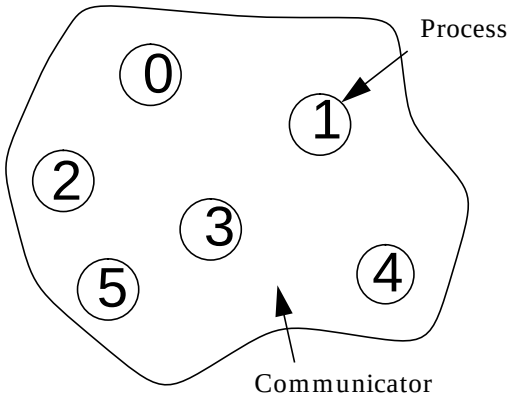

Figure 5: A Communicator

All message passing libraries have the concept of an address for a particular process, but MPI is unusual in having both a communicator and a rank for an address. Why do we need to specify two pieces of information where only one is sufficient for most libraries?

3.4 Library Construction

As all programmers will know libraries of pre-built functions which provide a toolbox of facilities are extremely useful. The concept of building a set of functions which perform generally useful operations (such as the NAG maths library) helps make programming slightly easier.

Unfortunately there are very few libraries at the moment which provide high level facilities, and which can operate on a parallel computer using message passing. The reason for this is primarily the difficulties of constructing a library whose internal operations do not interfere with the application.

As we will see user applications which employ message passing achieve all their coordination by exchanging messages. In theory any useful library would also achieve its work by also exchanging messages. However if a library is to be useful to the application it is important that library messages are kept separate from application messages.

In most message passing libraries this is impossible, as the library and application simply specify an address, and the destination process reads using a single function, i.e., the read function has difficulty in distinguishing between messages meant for it, and messages meant for a library (or more strictly the internal workings of the library).

In MPI this difficulty is solved by allowing libraries to specify their own communicators, within which their messages pass. As the application will use a different communicator it need not be aware of any other messages being transferred. In other words the read function is clever enough to be able to differentiate.

This, then, is the reason that the MPI programmer has to specify both the communicator and the rank in a message transfer!

4 Programming in MPI

This chapter introduces the student to actually programming MPI.

4.1 Starting MPI

The first MPI function any program must make is the MPI_INIT call. The C and Fortran versions differ slightly, as it is common in C applications to pass command line arguments to libraries, but not in Fortran.

The C binding looks like:

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

and the Fortran binding is:

```
MPI_INIT(IERROR)
```

where IERROR is an INTEGER.

This call allows the MPI implementation to perform any necessary bookwork to ensure future function calls work. Obviously exactly what is performed varies between implementations.

4.2 Stopping MPI

After you program has done all the necessary work it must tell the MPI implementation to shut down *before the program terminates.*

This allows an MPI implementation to tidy up memory, and complete any outstanding operations. Confusingly in some implementations it may be possible to not shut down MPI, but such programs are not portable. There are two ways in which MPI can be halted, either gracefully or aborted.

The graceful shut down should be used when everything has gone to plan, and we are about to terminate the program. In this case a call to MPI_FINALIZE is made, which in C looks like:

```
int MPI_Finalize();
```

and in Fortran is

```
MPI_FINALIZE(code)
```

where code is an integer which tells us whether we've successfully terminated MPI.

If, however, something goes wrong in our application we may wish to shut down MPI immediately. This 'panic' call will force all processes in a particular communicator to

stop executing now, as something has occurred which meant that continuing was pointless. This is the MPI_ABORT call, which in C looks like:

```
int MPI_Abort(MPI_Comm comm, int errorcode)
```

This takes a communicator, and an errorcode which will be passed to the applications which are terminating.

4.3 Communicators

As discussed in Figure 2.1 most MPI programs conform to the SPMD model, i.e., at run time the program determines what its going to do. We have already mentioned that this decision is often taken on *where* the program is executing, and as MPI can only give location information in terms of communicators it is essential to look at how a program can obtain this information.

MPI provides two functions which are pertinent to this, the first of which obtains the rank of the process within a particular communicator;

```
int MPI_Comm_rank(MPI_Comm comm, int *rank);
```

This will set `rank` to the rank of this process in the named communicator.

The second routine returns the number of processes currently executing within the named communicator;

```
int MPI_Comm_size(MPI_Comm comm, int *size);
```

Together this functions are normally used in the initial SPMD code to determine which routine to execute.

4.4 Writing MPI Programs

We have now covered the extreme basics of MPI programming, and with the simple MPI calls so far described it is possible to produce a useful program (as we will see shortly). However we need to know one more piece of information to write an MPI application - how we link in the library.

MPI, as previously mentioned, is a document, and the details of how you *link* your application to a particular implementation is not standardised. However the standards document does specify what header files must be included within your program.

These files prototype the various MPI functions and datatypes, and should be included within any MPI source program in much the same way you might include any libraries header files.

The C file is linked with

```
#include "mpi.h"
```

and the Fortran file with

```
include 'mpif.h'
```

4.5 Exercise 1

We have now covered the basics of the concepts and design of MPI, and to reinforce this we now

Write a program which uses MPI to start 2 child processes, each of which prints a "Hello" message to the screen, then exits.

This basic application should introduce the student to using MPI.

This program can be completed using the functions described in the previous chapters, which are summarised in Table 1 & Table 2.

Table 1: Basic MPI Functions (C Bindings)

C Function	Description
int **MPI_Init**(int *argc, char **argv);	Initialise the MPI Implementation
int **MPI_Finalize**(void);	Gracefully shut down the MPI implementation
int **MPI_Abort**(MPI_Comm comm, int errorcode);	Attempt to shut down all processes in named communicator.
int **MPI_Comm_rank**(MPI_Comm comm, int *rank);	Set the rank of the calling process in named communicator.
int **MPI_Comm_size**(MPI_Comm comm, int *size);	Set size to number of processes in named communicator.

Table 2: Basic MPI Functions (Fortran77 Bindings)

Fortran Function	Description
MPI_INIT(IERROR) INTEGER IERROR	Initialise the MPI Implementation
MPI_FINALIZE(IERROR) INTEGER IERROR	Gracefully shut down the MPI implementation
MPI_ABORT(COMM, ERRCODE, IERROR) INTEGER COMM, ERRCODE, IERROR	Attempt to shut down all processes in named communicator.
MPI_COMM_RANK(COMM, RANK, IERROR) INTEGER COMM,RANK,IERROR	Set the rank of the calling process in named communicator.
MPI_COMM_SIZE(COMM, SIZE, IERROR) INTEGER COMM, SIZE,IERROR	Set size t number of processes in named communicator.

4.6 A Note on `mpich`

This course is intended to be taught using one of the major MPI PD implementations, mpich.

All of these implementations use their own way of causing the execution of applications, and mpich tackles this problem by providing a 'launcher' application.

When your application is compiled you will run it by typing:

```
mpirun -n <number of processors> <progname>
```

5 Message Passing in MPI

In order for MPI to help an application developer in the production of a message passing application it must provide a higher level interface to the low level data transmission primitives modern OS's provide. MPI does this by providing a large number of message passing functions, which together allow the application developer a great deal of flexibility.

This chapter of the notes looks at how these functions may be used to write basic message passing applications.

5.1 A Message

The basic OS data transmission primitives tend to view a message as a chunk of memory, which will be copied to the destination machine, and many early message passing libraries (notably PVM) adopt this same approach. MPI, however, takes a different view of the operation.

An MPI implementation can only send a message if its told what is in the message.

This rather bland statement has a large effect on the way in which MPI applications are constructed, as will become clear in this and the next chapter. For now we'll look at what a message looks like.

In MPI a message is an array of elements, all of some previously defined datatype. When we come to send a message, (or receive one), we must tell MPI what is within the message. So, for example, Figure 6 shows a very simple MPI message we might wish to send, which just consists of 5 integers.

Before we look at how we tell MPI what goes within a message we should first look at what datatypes MPI knows about. In practice the two different language bindings (C & Fortran) define two sets of datatypes, i.e., those which correspond to the datatypes used in those languages.

The datatypes which MPI understands are summarised in Table 3 & Table 4. As you can see there is some overlap (they both know about integers for example).

Why go to all this trouble? Other message passing libraries seem to perform adequately where they are simply moving memory around. In practice there are two answers to this question:

- Application maintenance, and,
- Inter-machine message portability.

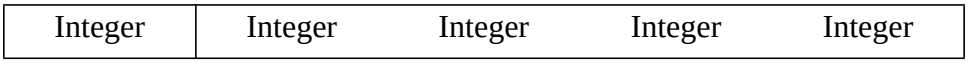

Figure 6: A Basic Message

Table 3: MPI Datatypes (C Binding)

MPI Datatype	C Equivalent
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	char
MPI_PACKED	

Table 4: MPI Datatypes (Fortran Binding)

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

We will discuss the first reason at some length in the next chapter, and for now will concentrate on what “inter machine message portability” means.

Most programmers are familiar with the concept of using floats, complex numbers and structures in their programs, and these concepts are portable between different compilers. However, anyone who has performed any low level programming will also be aware that the processor used by the computer usually uses its own way of encoding variables into memory.

Computer memory simply consists of a sequenced collection of words (where the length of the word (in bits) varies between processors), which the processor manipulates. The mapping between chars in C (for example) and words is usually trivial (i.e., a char is placed in a single byte), but the way in which we encode other variable types (in particular floating point numbers) is usually far more complicated.

In normal computer programming we don't need to care about how our floating point numbers are encoded into memory, after all the compiler does all the hard work, but in parallel programming it becomes an issue, because *different processors will use different ways of encoding these variables.*

This means that if we place a floating point number in a piece of memory on a HP workstation, then copy that memory to a SUN workstation, there is no guarantee that the SUN will be able to interpret the same number from that sequence of words.

This is a common problem in message passing, and is normally worked around by adding an extra step to the data transmission process, where the data to be transmitted is converted to an intermediate format before transmission, and then on receipt is converted into some machine specific form. For example PVM uses SUN's XDR library to do exactly this.

Unfortunately most MP libraries require the programmer to tell the library whether this transformation needs to take place, for example a PVM programmer must initialise buffers to use this transformation if the buffer will ever be sent to a different machine. This

- makes the programmers life harder, and
- may mean that more transformation will take place than is strictly needed.

MPI attempts to solve this problem by requiring the programmer to tell it what type of data is being sent to the other machine, i.e., the message contents. Then, at send time, the MPI implementation can determine which portions of the message need transforming, without the programmer needing to be aware of this.

In other words, the MPI implementation will take on the responsibility of ensuring that the message can be interpreted by the destination machine, whatever its internal processor datatypes.

The result of all this is that MPI's insistence on knowing what is in a message may initially seem like more work, but potentially could lead to faster applications, and less effort for the programmer.

5.2 Constructing Messages

As we have seen MPI messages are assumed to be a sequence of datatypes, *all of the same type*. This makes MPI particularly good at transmitting arrays of data, but for anything else it seems horribly restrictive!

Does this mean that in order to send messages of different types you must send separate messages? No, but we will not discuss how we do this until the next chapter. For now remember that although we are talking about a very restrictive form of messages we will loosen that restriction later on!

5.3 Point to Point Communication

There are various ways of sending messages between processes, but by far the most common is point to point communication, i.e., a message travels from a single source to a single destination.

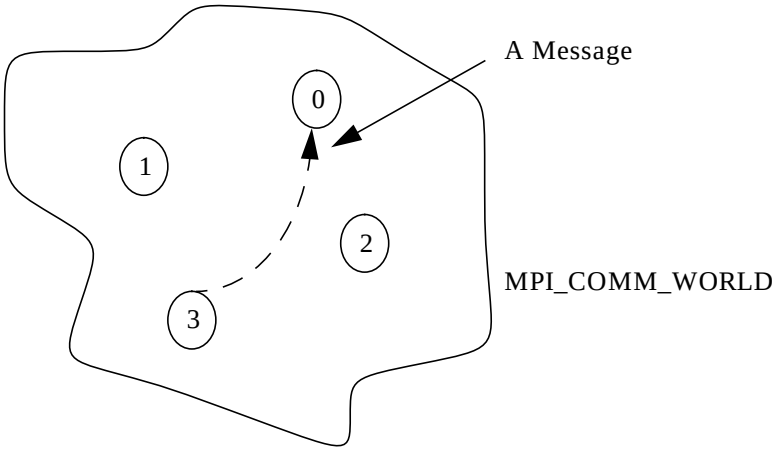

Figure 7: Point to Point transmission

We have already seen that messages travel within a particular communicator, and so we can view the point to point comms style as shown in Figure 7, where a message travels from a process with rank 3 to one with rank 0 in MPI_COMM_WORLD.

In practice an application writer requires greater control than simply “it goes from here to here”, and MPI provides 4 communication modes:

- Synchronous send
- Buffered Send
- Standard Send
- Ready Send

We will now discuss each of these in turn, and attempt to identify why you might want to use each mode in particular situations.

5.4 Synchronous Send

Quite often when we are writing distributed applications we need to know that the destination process has received a message and *is acting upon it* before we can proceed.

We cannot really do this without being sure that the destination process received a particular message we sent it, and the application itself (not just a buffer somewhere) has received it.

Before we can discuss how we MPI can achieve this we must first review some of the basics of message transmission.

5.4.1 Data Transmission.

A large variety of networking technologies exist (ATM, Ethernet, FDDI, Token Ring etc)., all of which were designed for particular situations. Whilst all message passing libraries try to hide the specific details of these low level implementations there are always some details the programmer needs to be aware of.

The most important detail the programmer needs to be aware of is the existence of *buffering*.

As most networks cannot guarantee a particular level of service, or speed, they have to do some work to enable faster pieces of technology to work with slower units. They do this by buffering data.

This means that when a piece of memory (or data) is transmitted over the network it isn't copied directly from the computer memory across the network. In practice it is copied first into a buffer. When the time comes for an actual transmission, or when the network is ready, the data is copied *from this buffer* onto the network.

This means that the message transmission (at the lowest level) tends to be split into 4 steps:

- Source application places data into the outgoing buffer.
- The networking system (software or hardware) arranges for the data in this outgoing buffer to be transmitted across the network when it can.
- The destination processes networking subsystem receives the data and places it in its local incoming buffer.
- The destination application copies the data from its incoming buffer into its own memory.

After the first step the source process has done its work, from then on the timing involved may be dependent on a number of factors, principally the congestion of the network, and whether the destination process is ready to receive data from its buffer into its memory (the last step).

All the MPI message passing functions sit on top of this (somewhat idealised) version of the message transmission process.

5.4.2 Back to Synchronous Send

The MPI synchronous send call hides the buffering between the two applications, and will only complete when the last of the 4 stages has been completed, i.e., when the destination application has copied the data out of its incoming buffer into its memory.

A common way of looking at this is via a time chart, as shown in Figure 8. Here we show two processes. At the beginning both are performing useful work, then the source process makes a synchronous send call to the destination. For some reason the transfer isn't instantaneous (message transmission very rarely is in practice!) in this case the destination process is still doing some useful work, and so the final stage of the process (the copy from the destinations incoming buffer) is delayed.

Meanwhile the source process has to wait (indicated by the dashed line). Eventually though the destination process does receive the data, and at that point the source process can carry on.

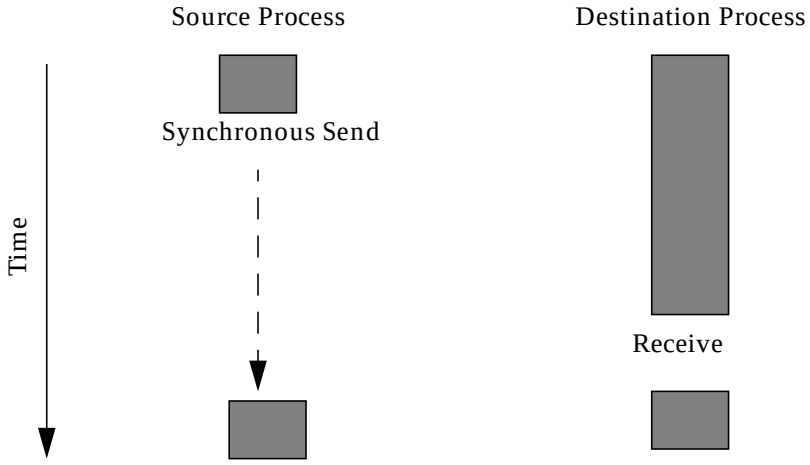

Figure 8: A Synchronous Send time chart

In MPI the C binding for the synchronous send call looks like:

```
int MPI_Ssend(buf, count, datatype, dest, tag, comm)
```

As this is the first time we've seen an MPI send call its worth looking closely at the parameters we pass, as they remain largely similar between comms. modes.

- `buf` is the address at which the message starts in the application.
- `count` is the number of datatypes, of type `datatype`, which is in `buf` which we wish to send.
- `dest` is the rank of the destination within the communicator `comm`.
- `tag` is an application specific identifier which we associate with the message, we'll talk about this more when we discuss receives.

If we were sending the array of integers (in C) which we showed in Figure 6 then the excerpt of code would look like:

```
int int_array[5];

/* Put data in the array */

/* Now send the message */
MPI_Ssend(&int_array[0],
        5,
        MPI_INT,
        3,            /* We're sending to rank 3 */
        0,            /* Tag of 0 */
        MPI_COMM_WORLD);
```

This function will only return when the process of rank 3 in MPI_COMM_WORLD receives the data from its internal buffer.

5.5 Buffered Sends

As we have seen synchronous sends can cause performance problems (the source process in Figure 8 spent a lot of time waiting for the message to be received) and in most cases we don't want to put up with this.

We would like to do some useful work while waiting for the message to be sent, and synchronous send can't let us do this, as the function call won't return until the transmission is thoroughly complete.

Buffered sends allow us to do this by adding an extra buffer between the application memory and the network buffer. Then when the network comes around to performing the send it will copy the data from this application buffer into the network buffer, and then transmit it over the network.

The process (as shown in Figure 9) consists of copying the data from the application memory (i,e, the address given as the buf parameter in the function call) into an application buffer. The buffered send call terminates once the data has been copied into the application buffer, and so the source process can do some useful work.

Before we look closely at the actual buffered send calls we must first look at how the application buffer is managed.

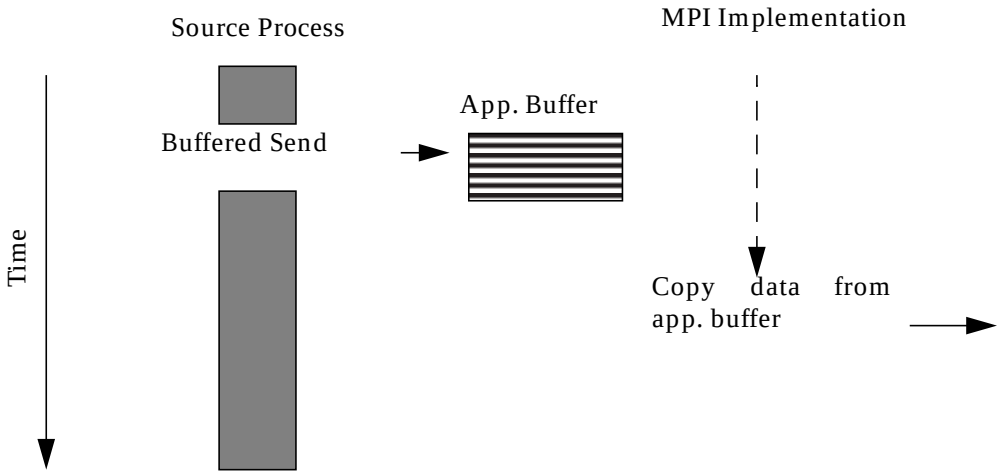

Figure 9: Buffered Sending

5.5.1 Application Buffers

MPI allows only one active application buffer at any one time. This buffer is simply a piece of memory which has been allocated by the application, which it wishes to pass over to the MPI implementation for its use.

Once the memory has been allocated (using malloc() in C or via an array in Fortran) we tell MPI to use it by calling

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

This tells MPI to use the space of `size` bytes (or elements in Fortran) starting at address `buffer` as the application buffer for subsequent buffered sends.

Once we have finished, or we wish to change the buffer in some way, perhaps to make it large enough to take other messages, we must tell MPI to stop using it.

We achieve this by calling

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

This tells MPI to no longer use this buffer, BUT doesn't free it from the applications use. It can continue using that memory until it is either freed (in C) or falls out of scope.

5.5.2 Buffered Send

Once we have allocated a sufficiently large buffer for our buffered sends we can initiate a buffered send call. This looks like:

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

As you can see the function parameters are identical to those used in synchronous sends, the only difference being the B in the function name.

5.6 Other Communication Modes

We have now looked at the most important communication modes, synchronous and buffered sends, and in practice most MPI programs will be written using these two modes.

There are two further modes which we will mention here, though we will not cover them in much detail.

5.6.1 Standard Mode

MPI provides a less straightforward comms mode called standard send. This send will complete once the message has been sent out of the source machine, which may or may not imply that the message has arrived at its destination.

Internally an MPI implementation may use buffered or synchronous sends.

5.6.2 Ready Sends

All the comms. modes we have talked about ensure that the message will eventually reach the destination. In some very rare circumstances we can relax this restriction, and infer whether a message transfer was successful through other means.

Ready sends will be successful if the destination process “can take the message”. These sends complete immediately, and will be successful if the destination process is already waiting for a message.

If the destination isn’t waiting, then what happens isn’t defined by the standards document. Most MPI implementations take this to mean that the message will be dropped, but the sender isn’t told this.

This unusual form of communication is useful in some very rare circumstances, but most MPI programmers will never need to make use of it.

5.7 Receiving in MPI

We have now discussed how an MPI program can send a message to a destination process, and we must now address how the destination process receives an incoming message.

The most common form of receiving is called the “blocking receive”. This means that when we call the blocking receive function, the program will not carry on until the desired message has arrived.

Before we look at the function call it is worth discussing how the function specifies the message it wants to wait for.

5.7.1 Tags

When we first introduced sends (Section 4.4.2) we mentioned the concepts of a tag, which is an application specific number which we associate with a particular message.

More strictly, a tag is a piece of information which is associated with a message, but which isn’t contained within it. The most apt real world example is that of an envelope, which often contains labels on it to indicate what is within the envelope before you open it.

In much the same way envelopes enable you to classify your post, tags in MPI allow programs to chose what type of message it is to receive. This is particularly important in MPI, as the receive operation must specify the contents of the message in the receive call, if the wrong type is specified, then an error may occur. Therefore it is critical for an MPI program to be able to request receipt of a particular type of message.

In MPI, like most message passing libraries, incoming messages are buffered. We have already said that there is an 'incoming' buffer, but in actual fact the software goes to some lengths to make this single buffer look like several buffers.

The incoming buffer appears as a collection of queues of messages which have been received, but which the application has yet to transfer into its memory. So for example the real world model of an incoming buffer might look like the one shown in Figure 10.

When we actually come to execute a receive call in MPI the implementation looks in these queues of outstanding messages to decide which message to return to the application.

If that call specifies a particular tag then MPI will look for the queue corresponding to that tag (for example Bills in the example below). If a message of that type is there, then the first one is returned.

An MPI program can also specify the source of the message, so for example you might only wish to read your gas bill. In this case the MPI implementation would search the relevant queue for the first message from that source.

Alternatively the receive call can be more lax, and just ask for a particular source, of any tag, and so MPI will search all the queues looking for the first message from that source.

It should be clear then that the tags associated with messages are very important in MPI, and it is critical (at least in large applications) to pick meaningful tags. As in MPI these tags are simply integers, it is common to define meaningful names which make reading the programs more pleasant.

5.7.2 The Blocking Receive

The blocking receive call looks like:

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

These parameters are worthy of further discussion.

- This call will return a message of tag tag, which came from the process of rank `source` in the `comm`.
- The message will consist of `count` instances of type `datatype`, and will be placed in the memory starting at address `buf`.

The tag and source parameters can be integers to enable the program to be very fussy in the message it receives, but they can also be special parameters to allow wildcard matches.

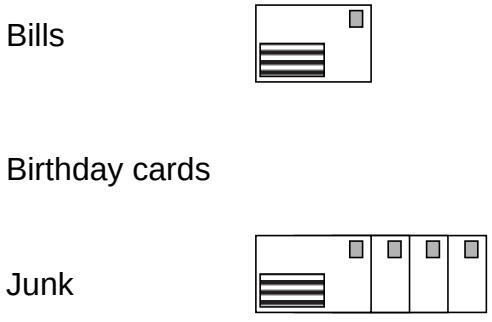

Figure 10: Real world incoming buffers

If the source parameter is MPI_ANY_SOURCE, then a message from any process is considered, and likewise if the tag parameter is MPI_ANY_TAG then a message with any tag is considered.

The status parameter is the last item of note.

If an application chooses to receive a wildcard type of message (i.e., not be too choosy) it may wish to find out afterwards the source or tag of the message. MPI allows this by setting a status structure with this information.

In C this is a structure, the contents of which are

```
status.MPI_SOURCE; /* Rank of the source process */
status.MPI_TAG; /* Tag of this message */
```

This structure is of type `MPI_Status`. In Fortran the equivalent information is placed in a 2 item INTEGER array (which you pass to the receive call). The items within the array are

```
status(MPI_SOURCE)
status(MPI_TAG)
```

5.8 An Example

To illustrate how the send and receive processes work together consider the two programs shown in Figure 11. Here we have two processes running, one of which (the source process) is executing on a process other than rank 0, and it wants to send the floating point number 10.0 to the rank 0.

It does this by calling a Synchronous send, and the message transfers inside MPI_COMM_WORLD. Meanwhile the recipient process (rank 0) doesn't know which processor the next message is coming from, or its tag, but it knows that it will contain 1 floating point number.

It receives the message into result using a blocking receive, and then uses the my_status variable to find where it came from.

```
float sum = 10.0;
MPI_Ssend(&sum, 1, MPI_FLOAT, 0, 1,
          MPI_COMM_WORLD);
```

Sending Process

```
float result;
int result_from;
MPI_Status my_status;

MPI_Recv(&result, 1, MPI_FLOAT, MPI_ANY_SOURCE,
        MPI_ANY_TAG, MPI_COMM_WORLD,
        &my_status);
result_from = my_status.MPI_SOURCE;
```

Receiving Process

Figure 11: Example Message Passing

5.9 Note on Wildcard Receives

Wildcard receives allow us a great deal of flexibility when constructing our application, but it is worth remembering that we must still specify the contents correctly. If we specify the wrong contents the MPI implementation may have problems.

Therefore we should only use wildcards where all the messages could fit in the same buffer using the same datatype....which is very restrictive.

5.10 Non-Blocking Communications

We have now talked about the main MPI message communication primitives; namely the 4 blocking communication modes, and the blocking receive call. These calls are all called *blocking* calls because they don't terminate (i.e., the calling program cannot carry on) until they have completed the work they do.

As we have seen if the communication takes a long time to perform this will cause performance penalties, though buffered communication may help in these circumstances.

However buffered communication suffers from two problems:

- The user must manage the buffers, and
- It takes some time to copy data into the application buffer

To address these problems MPI provides a while suite on *non-blocking calls*, which are the topic of the rest of this chapter.

5.10.1 Non-Blocking Theory

A non-blocking communication in MPI is a two-step process; an initiate communication and test for completion. The first call begins either the send or the receive, and will complete immediately. After this it is possible to perform some useful work, while the communication is still underway.Then, when we can no longer perform any more work we can check whether the communication has completed.

There are 3 non-blocking send modes in MPI, 1 non-blocking receive call, and a large number of test calls for the second stage.

5.11 Non-Blocking Sends

5.11.1 N.B. Sync Sends

The non-blocking synchronous send call looks like

```
MPI_ISsend(buf, count, datatype, tag, comm, request)
```

The parameters are identical to the blocking send version, except for the addition of a request parameter. The request parameter in C is of type MPI_Request, and in Fortran is of type INTEGER.

This request type is a token than MPI attaches to this particular message. Later, when we wish to find out whether a particular communication has completed we will use this request to indicate the comm. we are talking about.

5.11.2 Other N.B. Sends

There are non-blocking forms of standard and ready sends, but as these are not used by most MPI programmers we will gloss over them in this introductory course.

5.12 Non-Blocking Receives

In some environments the receive operation may take some time to perform, and so MPI provides a non-blocking form of receipt. The non-blocking (or immediate completion as some portions of the MPI standard describe it) receive looks like

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

Note that again the only difference between blocking and non-blocking receives is the function name and the addition of the request parameter.

5.13 Testing for Completion

We have seen how we initiate non-blocking sends and receives, but what about testing for completion?

In MPI there are two completion modes:

- Waiting for completion (i.e., block until the relevant communication is complete)
- Test for completion, and return with the information immediately.

For single communications these modes take the form of

```
MPI_Wait(request, status)
```

and

```
MPI_Test(request, flag, status)
```

The first function will wait until the named comms (in the request) is complete, and the `status` parameter will be set to whatever would normally happen in a blocking call. The second function will set the `flag` to TRUE or FALSE depending on whether the comm is complete.

Why have a status field in the test for a non-blocking send? In the interests of reducing the number of functions in MPI the status field is normally ignored by MPI applications if it pertains to the completion of a *send*.

5.13.1 More Advanced Completion Tests

MPI also provides completion tests for collections of outstanding communications, which operate by taking arrays of requests, and setting arrays of status parameters.

These functions include:

- MPI_WAITANY
- MPI_WAITALL
- MPI_WAITSOME
- MPI_TESTANY
- MPI_TESTALL

- MPI_TESTSOME

The ANY variant will return information about the first item of interest in the array. The ALL variant will block until all have succeeded, or return information about all the outstanding comm.s Finally the SOME variant is similar to ANY but instead of only returning information about the first will return information about as many as possible.

5.14 Basic Message Functions

Table 5: Message Functions (C Binding)

Function Name	Operation
int **MPI_Ssend**(void *buf, int count, MPI_Datatype datatype , int dest, int tag, MPI_Comm comm)	Blocking synchronous send
int **MPI_Buffer_attach**(void *buffer, int size)	Tell MPI to use this buffer in buffered sends
int **MPI_Buffer_detach**(void **buffer, int *size)	Stop using this buffer
int **MPI_Bsend**(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)	Buffered Send
int **MPI_Recv**(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)	Blocking receive
int **MPI_Issend**(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)	Initiate non-blocking synchronous send
int **MPI_Irecv**(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request	Initiate non-blocking receive
int **MPI_Wait**(MPI_Request *request, MPI_Status *status)	Wait until named communication complete
int **MPI_Test**(MPI_Request *request, int *flag, MPI_Status *status);	Test to see whether named communication complete.

Table 6: Message Functions (Fortran Bindings)

Function Name	Operation
MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR) <type> BUF(*)	Blocking Synchronous Send
MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR) <type> BUF(*)	Tell MPI to use this buffer in buffered sends
MPI_BUFFER_DETACH(BUFFER, SIZE, IERROR) <type> BUF(*)	Stop using this buffer
MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)	Buffered send

Table 6: Message Functions (Fortran Bindings)

Function Name	Operation
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR) <type> BUF(*) INTEGER STATUS(2)	Non-blocking receive
MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) <type> BUF(*)	Initiate non-blocking synchronous send
MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG COMM, REQUEST, IERROR) <type> BUF(*)	Initiate non-blocking receive.
MPI_WAIT(REQUEST, STATUS, IERROR) INTEGER STATUS(2)	Wait for named completion
MPI_TEST(REQUEST, FLAG, STATUS, IERROR) LOGICAL FLAG INTEGER STATUS(2)	Test for named completion.

Note that all non-specified parameters in the Fortran bindings are INTEGERS.

6 Derived Datatypes

We have now discussed the basics of message transfer using MPI. As we mentioned earlier MPI assumes the message consists of a sequence of identical datatypes, which somewhat restricts the sort of messages we might wish to send.

MPI allows us to work around this by letting us define our own new application meaningful datatypes. In practice this has two advantages:

- It allows the MPI implementation to make sensible decisions on what needs to be encoded for transport between non-identical processors.
- It also makes reading and maintaining large message passing programs easier, as it is possible to see what is being passed around in an application context.

Unfortunately the disadvantage of this is that there is a certain amount of effort involved in defining the datatypes. However, on the whole, the necessity of deriving datatypes is more a help than a hindrance.

6.1 Background

Consider the following simple C datastructure (which we will use later on in the second exercise)

```
typedef struct work_packet
{
   int num_strips;
   float start_value;
} work_packet;
```

We wish to be able to send this structure between processes using MPI. Using the routines we saw in the previous chapter we cannot do this, as the two fields are of different types.

We could send two messages (one with a single integer and another with a single float), but this causes problems in ensuring the recipient gets them one after another.

This is exactly the situation MPI's derived datatypes feature is intended to address.

6.2 What is a derived datatype?

We have seen examples of describing a message using the intrinsic datatypes which MPI already knows about (MPI_FLOAT, MPI_LOGICAL and so on). When we derive a datatype we tell MPI about this new type, and then we can continue to use it as though it was an intrinsic type.

This new datatype is called derived because we describe it in terms of existing datatypes. So, in much the same way we described the work_packet structure in terms of C's ints & floats we describe the MPI datatype in terms of MPI_INTEGERS and MPI_FLOATS.

Whilst it is easy to see how this makes life easier when programming in C, and we wish to move struct's around, the derived datatype feature also allows variables which are not connected together in the source language (such as in Fortran) to be connected when we wish to transmit them.

It does this by generalizing datatypes to be patterns of memory, which are organised in a particular way, which is used to extract the datatype when we wish to send or receive it.This generality helps make it a useful feature in Fortran, but makes the process of describing derived datatypes unnecessarily arcane.

One of the weaknesses of the derived datatype technique is that, at first sight at least, it seems far more complicated than it needs to be.

The advantage is the way in which derived datatypes can be used to perform quite complicated operations.

With that warning we can now consider how we derive datatypes, but first a short digression.

6.3 Variables in Memory

In Section 4.1 we discussed how different processors encoded variables like floating point numbers into computer memory. Recall that computer memory consists of a sequence of words, each of which consists of a number of bytes (the exact number varying from processor to process).In essence they achieve this by splitting the number into a number of computer words, in an agreed format.

Then when they come to unpack the variable from memory they use the reverse process to find the number. In practice the process is complicated slightly as many processors have difficulty in reading memory which doesn't start on a word boundary.

Consequently two pieces of information are needed for this encoding:

- The number of bytes taken up by the data
- The offset from the start of where we put the data to the byte where we can start placing the next variable.

As we have mentioned the normal low level variable types (floats and so on) already have this information defined by the processor manufacturer, and all low level software uses this to find data.

Why, you may ask, are we talking about this very low level of detail? Because this process of defining the number of bytes and offset is *exactly the same process* we use when we derive a datatype in MPI.

6.4 Making a new type

When we derive a datatype in MPI we do two things:

Firstly we build up a description of the datatype in a format that MPI will be able to understand later on. This description is called a type map, and contains the two pieces of information we talked about in the previous section.

The second step involves passing this information to MPI. MPI then adds this information to its list of recognised types, and from then on you can use it as though it was one of the intrinsic types.

Finally, after your application has completed you can remove the datatype from MPI's list of accepted types, which is usually called freeing the type.

6.5 Type Maps

A type map is the stencil that MPI will use when it wishes to extract, or write, a structure of the type you have defined. Essentially it contains two columns of information, and as many rows as there are pieces of data in your datatype (this isn't strictly true, as we will see later, but is a workable approximation).

The two columns of the type map are:

- The basic datatype of this component (e.g., MPI_FLOAT)
- The Displacement of this component from the beginning address of the datatype.

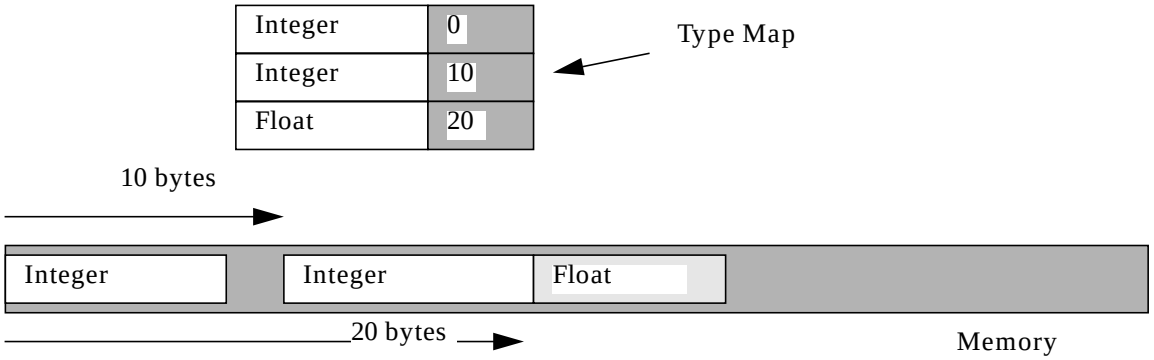

Figure 12: A type map as a stencil into memory

When we refer to a variable of this type once MPI has defined it, we always supply the start address. Then to find the relevant items within the derived datatypes MPI uses the displacements from that start address to find the relevant individual component datatypes.

In the example shown in Figure 12 the type map contains 2 integers and a float. The first integer is at the beginning of memory, the second 10 bytes from there, and the float 20 bytes from the start.

As you can see the type map concept is quite general, and as the variables are only related by their offset in memory they do not have to be connected as far as the language is concerned. For example, as we shall see, type maps can be used to extract particular items of interest out of arrays.

Having discussed the background to how derived types are managed in MPI we can now look at how we derive a datatype.

6.6 Deriving a Structure

In this section we will see how we can tell MPI about a structure, which to MPI means a collection of heterogenous variables. First it is worth looking at where we obtain the data to fill in the type map.

The first field (the type) is trivial, but how do we find byte offsets? Rather than requiring the programmer to have an intimate knowledge of the byte offsets used by the machine on which he or she is working MPI provide functions which allow the programmer to calculate such information in a form suitable for passing to MPI.

Awkwardly this may (the standard isn't clear) on whether this sort of information differs from that supplied by sizeof() in C, so the C programmer must put up with a new way of finding information he or she is quite familiar with.

We find the size of a piece of data using the function

```
MPI_TYPE_EXTENT(datatype, extent)
```

This places the length of the datatype in the variable extent.

Back in Section 5.3 we saw that the low level internal intrinsics usually have some offset after the data to the next word boundary where we can place information. The MPI extent function calculates the actual length of the data, and adds onto it this offset.

In otherwords the extent returned by this is the number of bytes after the start of this data at which the next datatype will be placed. So in the example shown in Figure 12, when we wish to find where the second integer is placed, we call MPI_TYPE_EXTENT with an integer to find how much space the earlier variable took up (in this case 10 bytes).

We can now begin to consider how we derive a datatype. As an example we will use the structure we mentioned earlier:

```
typedef struct work_packet
{
   int num_strips;
   float start_value;
} work_packet;
```

Rather than constructing the type map explicitly we build it by calling MPI functions.

6.6.1 Datatype Column

The first of which we will consider is the left hand column, or the fields. We prepare this as an array of MPI_Datatypes, (or INTEGERS in Fortran). Each item is set to the type which corresponds to that entry. For our work_packet structure a piece of code like:

```
MPI_Datatype array_of_types[2];
array_of_types[0] = MPI_INTEGER;
array_of_types[1] = MPI_FLOAT;
```

will do the trick. Remember that in C MPI's arrays start at 0, whereas they start at 1 if we are using Fortran.

6.6.2 Offset Column

The second column is that of the offsets from the start of the structure, which as we have seen we obtain by calling MPI_type_extent. We place these offsets within an array of MPI_Aint's (or INTEGERS in Fortran).

Note that the entries in this array are offsets from the start of the structure, in other words the size of the *previous entries*, not the current one. So the entry for component 3 is the sizes of components 1 & 2 added together.

In our example the following code excerpt does the trick:

```
int int_length;
MPI_Aint array_of_displacements[2];
```

```
struct {
   int count;
   float minimas[10];
   int tmp;
}
```

Figure 13: A more complicated structure

```
MPI_Type_extent(MPI_INT, &int_length);
array_of_displacements[0] = 0;
array_of_displacements[1] = int_length;
```

Given out simplified explanation of what constitutes a type map it now seems like we've generated enough information to describe our simple structure. In practice though MPI provides yet more generality; the ability to specify blocks of variables in one go.

6.6.3 Blocks

We have previously indicated that the entries in the arrays correspond to single entries in the structure that we are deriving. In fact this isn't quite true!

The designers of MPI felt that this simple two-column type map was sufficient for all cases, but was harder to work with for some circumstances than it should be. So they generalised the definition of each entry in the type map to a *block of variables*, rather than an individual variable.

To see why they felt this was necessary consider the structure shown in Figure 13. If we wished to describe this structure in our simplified type map we would have to fill in 12 separate entries for each of the 12 variables. The designers of MPI felt that, as MPI already dealt with arrays of like variables so well it should also be possible to place arrays in the components of the type array, so rather than saying

```
array_of_displacements[0] = 0;
array_of_displacements[1] = int_length;
array_of_displacements[2] = int_length + float_length;
array_of_displacements[3] = int_length + (2*float_length);

....
```

we would be able to tell MPI that the next n elements were all of the same type, and so it could calculate the displacements.

MPI achieves this by having a third array (which will go into the type map) which we call the blocklength array. In the example shown in Figure 13 this array would look like the one shown below.

Table 7: Blocklength array

Number of elements
1
10
1

For our earlier example (the work packet) this array is 2 elements long, and each element has the number 1 in it to tell MPI that each component in the other fields corresponds to a *single variable.*

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

6.6.4 Describing the Structure

Having now considered the different elements of the type description we can now tell MPI about this description. This stage simply allows MPI to take the 3 arrays we have constructed, and builds them into the type map that MPI itself uses.

This internal type map corresponds to an `MPI_Datatype` in C, and an INTEGER in Fortan. The C code to describe the structure for the example we have been building up looks like:

```
MPI_Datatype MyNewType;

MPI_TypeStruct(2,
               array_of_blocklengths,
               array_of_displacements,
               array_of_types,
               &MyNewType);
```

This fills in the MyNewType structure with the information MPI needs to be able to interpret the derived datatype in the future.

6.6.5 Committing the Datatype

Finally we need to tell MPI to begin accepted structures of this type. This is called committing the type, and is performed through the simple call:

```
MPI_Type_commit(&MyNewType);
```

6.6.6 Using the Datatype

Having described and committed the datatype we can now use is in send and receive calls as though it was an intrinsic datatype, for example;

```
work_packet tmp;

MPI_Ssend(&tmp, 1, MyNewType, 0, 1, MPI_COMM_WORLD);
```

will send the packet tmp to the 0th process, and likewise the receive call might look like;

```
work_packet incoming;

MPI_Recv(&incoming, 1, MyNewType, MPI_ANY_SOURCE, 1,
        MPI_COMM_WORLD);
```

6.7 Deriving Vectors

We have now seen how we can derive a structure for transmission in MPI messages. Whilst this is extremely useful MPI can derive datatypes for more complicated examples than simple structures. In this introductory course we will only detail one more

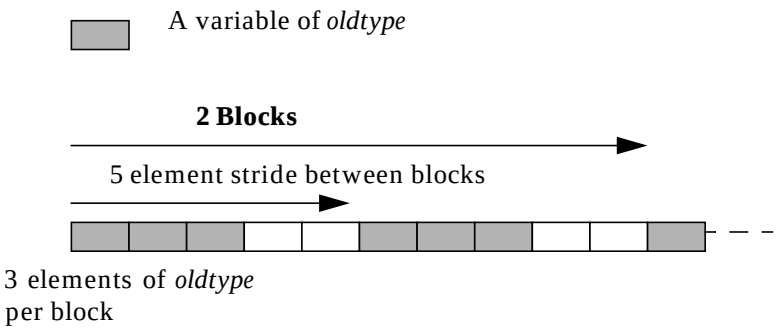

Figure 14: An MPI Vector

type, the vector, and the student is encouraged to refer to the reading list in the appendix for more types.

6.7.1 An MPI Vector

Imagine the case where we have a long sequence of data items stored in memory, such as a matrix, and you wish to extract certain items from it. As the normal MPI message assumes all the items are the same type the only way we could do this using simple calls is to copy the desired items into another array, which we would then send.

As this is a common operation (especially in fields such as matrix manipulation) MPI allows us to define a special derived datatype to do the extraction and send in one operation, hopefully very quickly. MPI calls these datatypes *vectors*.

An MPI vector draws its element from an array of variables all of the same type. As we are deriving a new type from this array we often call the component variables *oldtype*.

A vector consists of a number of blocks, in exactly the same way an MPI structure consists of a number of blocks (Section 5.6.3). Within each of the vector blocks there are

- A number of elements of oldtype that we wish to extract, and
- The offset from the start of the block to the beginning of the next one, called the *stride*.

If we set the stride to 0, then we are simply describing the array, if we increase the stride we are extracting out particular elements from the larger array.

In the example vector shown in Figure 14 there are 3 elements of oldtype that we are interested in, and 2 empty elements to the beginning of the next block (so this means a stride of 5).

6.7.2 Describing the Vector

As the vector definition is reasonably straightforward (at least in comparison to the structure) we don't need to build up any complicated arrays for the definition process.

Instead the description process is a simple call

```
MPI_Datatype MyVectorType;

MPI_Type_vector(2, /* The number of blocks */
```

```
   3, /* The elements per block */
5,. /* The Stride */
MPI_INT, /* The oldtype */
&MyVectorType);
```

Here we are assuming that we're extracting integers.

6.7.3 Committing the Vector

The commit process is exactly the same as for the structure, namely a simple call to

```
MPI_Type_commit(&MyVectorType);
```

From then on we can use the vector as though it was a pre-defined type. Note that, just as in the structure, when we come to do a send call, we supply a pointer to the larger array, and our new extracted vector, and the extraction will take place during the send.

6.8 Exercise

To illustrate how messages are sent and received in MPI we now set a very simple exercise, which calculates the area under a function by a Newton-Raphson approximation.

We supply most of the code, except for the message passing calls, which you will provide!

The skeleton program we provide contains 2 functions, master_process and worker_process. On start up the program determines which it is to execute. The master splits the range of the function up into a number of strips, and allocates each parcel of strips to a different processor.

It does this by placing the relevant information into a `work_packet` structure which it will send to the workers. We have already provided the datatype derivation code, so you can use it! Therefore the first thing you need to write is:

- Code to send a work packet to the worker.

After the master process has sent out all the work packets it waits until it can receive messages back from the worker containing the areas under that section of the function, in other words floating point numbers. Within the loop you must write

- Code to receive a floating point number from each worker.

The first thing each worker process does is receive the work packet from the master, so you must write the code to

- Receive a `work_packet`

The worker then performs the integration on that range, (using routines already written for you), and then wishes to return the area to the master. To do this you must write code to

- Send the floating point area back the master.

In summary, then, you write 4 pieces of code in the skeleton application provided. Where you need to write these segments are marked with

```
/* INSERT YOUR CODE HERE */

/* END INSERT */
```

6.9 Advanced Message Functions

Table 8: Advanced Message Functions (C Binding)

MPI Function	Description
int **MPI_Type_extent**(MPI_Datatype datatype, int *extent);	Set extent to the length of the named Datatype
int **MPI_Type_struct**(int count, int *array_of_blocklengths, MP_Aint *array_of_displacements, MPI_Datatype *array_of_types, MP_Datatype *new_type);	Define a structure derived datatype
int **MP_Type_commit**(MPI_Datatype datatype);	Tell MPI to accept this datatype in future
int **MPI_Type_free**(MPI_Datatype datatype);	Tell MPI to stop accepting this datatype
int **MPI_Type_vector**(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)	Define a vector derived datatype

Table 9: Advanced Message Functions (Fortran Binding)

MPI Function	Description
MPI_TYPE_EXTENT(DATATYPE, EXTENT, IERROR)	Set EXTENT to length of DATATYPE
MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES, NEWTYPE, IERROR) INTEGER (All arrays) (*)	Define a structure derived datatype
MPI_TYPE_COMMIT(DATATYPE, IERROR)	Tell MPI to accept this type in the future
MPI_TYPE_FREE(DATATYPE, IERROR)	Tell MPI to stop accepting this type
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)	Define a vector derived datatype

All non-specified parameters in the Fortran calls are INTEGERs.

7 Conclusions

This course has introduced the basic concepts of MPI, in particular:

- The concepts that a message passing library must provide
- The background to MPI's development.
- How MPI 'has benefited' from earlier implementations.
- The philosophy of programming in MPI (using communicators and datatypes)
- How applications are constructed using MPI (the SPMD model)
- How messages are sent and received in MPI.
- How we derive more advanced datatypes for our applications.

In this introductory course we have left out a great deal of detail, in particular the grouping operations MPI provide, global reduction, profiling & more advanced datatypes.

A Example MPI Programs

This appendix presents and discusses solutions to the exercises used in the MAN T&EC “Introduction to MPI” course. These simple programs are intended to show certain key abilities of MPI, and do not demonstrate its full functionality.

Exercise 1 illustrates how MPI allows the programmer to deal with the multiple execution of a single program on a distributed network of workstations. It illustrates that the spawning of processes is beyond the scope of MPI, and therefore any implementation must perform this in a non-standard way. It also demonstrates how an SPMD program can be constructed to perform different tasks on startup.

Exercise 2 is intended to show how programs operating with MPI may communicate with one another to enable the programmer to construct a parallel application. The student is provided, as part of the course, with a skeleton application which integrates a function.

The function is hard-coded into the slave program, and the integration is performed simply by approximating the area to a number of strips, with the number being fixed regardless of the number of processors used (simply so the student can verify that they obtain the same (inaccurate!) answer).

The student must produce the communication code, which uses MPI to

- pass the number of strips, and start parameter, from the master to the slaves
- receive this information on the slave
- return the area under the function for the range processed by each slave and,
- receive this information on the master.

We use a derived datatype in MPI to pass the work packet from the master, or root process, to the workers.

The programs included in this document have these sections completed, but it is important to emphasise the point that in most message passing applications the actual message passing code should be seen as distinct from the application specific processing.

A.a Exercise 1

This chapter presents and discussed the source code for a program which could be used as the ‘answer’ to the first exercise, namely a program which can be used as a number of remote processes.

A.b Makefile

This makefile illustrates the sort of issues which a particular implementation of MPI must handle. This particular makefile is specific to mpich, a public domain implementation of MPI.

```
1 # This is the makefile for the first example in the MAN T&EC Course
2 # "Introduction to MPI".
3
4 ALL: default
5
6 # This Makefile takes the form of the default makefiles for the ANL
7 # implementation of MPI, which is based on the p4 message passing
library.
8
9 # User configurable options
10
11# First we set the default architectre and communication channel.
12
13ARCH        = hpux
14COMM        = ch_p4
15
16# Then we set a variable to indicate where MPICH is ~currently~
installed.
17
18MPIR_HOME   = /afs/mcc.ac.gb/users/cgu/public/martin/mpich
19CC          = cc
20CLINKER     = cc
21CCC         =
22CCLINKER    = $(CCC)
23F77         = f77
24FLINKER     = f77
25AR          = ar crl
26RANLIB      = ranlib
27PROFILING   = $(PMPILIB)
28OPTFLAGS    = -g
29MPE_LIBS    =
30MPE_DIR     =
31LIB_PATH    = -L/afs/mcc.ac.gb/users/cgu/public/martin/mpich/lib/
hpux/ch_p4
32FLIB_PATH   = -Wl,-L,/afs/mcc.ac.gb/users/cgu/public/martin/mpich/
lib/hpux/ch_p4
33LIB_LIST    = -lmpi -lV3 -lU77
34MPE_GRAPH   =
35#
36INCLUDE_DIR =  -I$(MPIR_HOME)/include
37DEVICE      = ch_p4
38
39### End User configurable options ###
40
41CFLAGS  =  -Aa -D_POSIX_SOURCE -D_HPUX_SOURCE -DFORTRANNOUNDERSCORE
-DHAS_XDR=1 -DSTDC_HEADERS=1  -DHAVE_STDLIB_H=1 -DHAVE_SYSTEM=1 -
DHAVE_LONG_DOUBLE=1 $(OPTFLAGS) $(INCLUDE_DIR) -DMPI_$(ARCH)
42CFLAGSMPE = $(CFLAGS) -I$(MPE_DIR) $(MPE_GRAPH)
43CCFLAGS  = $(CFLAGS)
44#FFLAGS  = '-qdpc=e'
45FFLAGS  =  $(OPTFLAGS)
46MPILIB  = $(MPIR_HOME)/lib/$(ARCH)/$(COMM)/libmpi.a
47MPIPPLIB  = $(MPIR_HOME)/lib/$(ARCH)/$(COMM)/libmpi++.a
48LIBS    = $(LIB_PATH) $(LIB_LIST)
49FLIBS   = $(FLIB_PATH) $(LIB_LIST)
```

```
50LIBSPP  = $(MPIPPLIB) $(LIBS)
51EXECS   = fpi ex1
52
53# Finally, after setting any useful variables we perform the simple
compilation
54
55default: $(EXECS)
56
57all: default
58
59fex1: fex1.o $(MPIR_HOME)/include/mpif.h
60      $(FLINKER) $(OPTFLAGS) -o fex1 fex1.o $(FLIBS)
61
62ex1: ex1.o $(MPIR_HOME)/include/mpir.h $(MPILIB)
63      $(CLINKER) $(OPTFLAGS) -o ex1 ex1.o \
64      $(LIB_PATH) $(LIB_LIST) -lm
65
66.c.o:
67      $(CC) $(CFLAGS) -c $*.c
68.f.o:
69      $(F77) $(FFLAGS) -c $*.f
```

A.c Main program (C Version)

This is the C version of teh main program.

```
1 /* This is the sample answer for the first exercise in the MANTEC
2   course, "Introduction to MPI". If it is executing as the root process
3   then it prints a message saying so, otherwise it will print a message
4    indicating that it is a slave process */
5
6 #include "mpi.h"
7 #include <stdio.h>
8 #include <math.h>
9
10int main(argc,argv)
11int argc;
12char *argv[];
13{
14  int myid, numprocs;
15
16  /* First initialise the MPI system, which sets up the communicator */
17
18  MPI_Init(&argc,&argv);
19
20  /* Find out how many processors the user has created for our use */
21
22  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
23
24  /* and now obtain the rank of ~this~ process */
25
26  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
27
28  if (myid == 0)
29  {
30    /* I am the root process */
31    printf("I am the root process\n");
32  }
33  else
34  {
35    /* I am only a worker */
```

```
36    printf("I am worker %d out of %d\n", myid, (numprocs-1));
37  }
38
39  /* and shut down the MPI communicator */
40
41  MPI_Finalize();
42}
```

A.d Main Program (Fortran version)

The Fortran version of the first exercise is very similar, the only principal difference being the addition of error variables being passed to most MPI functions.

```
1 c This is the sample answer for the first exercise in the MANTEC
2 c "Introduction to MPI" course. If it is executing as the root process
3 c then it prints a message saying so, otherwise it will inform the
4 c user that it is a slave
5
6       program main
7
8       include 'mpif.h'
9
10       integer myid, numprocs, rc
11
12c First initialise the MPI system, which sets up the communicator
13
14       call MPI_INIT( ierr )
15
16c Find out how many processes the user has created for us
17
18       call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs , ierr)
19
20c Now obtain the rank of ~this~ process
21
22       call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr)
23
24c Now determine what to do
25
26       if(myid .eq. 0) then
27          print *, "I am the root process"
28       endif
29
30       if(myid .gt. 0) then
31          print *, "I am a worker"
32       endif
33
34c Shut down the MPI communicator
35
36       call MPI_FINALIZE(rc)
37
38       end
39
```

A.e Exercise 2

The second exercise of the course is intended to show how data may be passed between programs as a way of forcing an application logic across multiple processes.

As mentioned this application is intended to perform a simple integration under a function, using a Newton-Raphson approximation. Whilst their is one program, only one instance of this performs the master_process() function, the rest function as workers.

The master splits up the number of strips into bundles for each worker to integrate. It packs the information it is going to send into an MPI_struct, which it defines. Once the worker has received this information it performs the approximated integration, then sends a message back to the worker containing the area under that parameter range.

The worker accumulates these values, adds them together, and prints the result to the user.

A.f Program (C version)

```
/* This is the sample answer for the second exercise in the MANTEC
  course, "Introduction to MPI". If it is executing as the root process
   then it prints a message saying how many slaves it has to work with,
    then uses them to calculate the area under the curve */

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

/* Prototypes for functions */

float integrate_strip(float start_value, float width);
float function(float value);
void master_process();
void worker_process();

/* and the definition of the work_packet which will be sent to the
   workers via MPI calls */

typedef struct work_packet
{
  int num_strips;
  float start_value;
} work_packet;
MPI_Datatype workDataPacketType;
```

This is the structure which will hold the work packet we send to the workers.

```
/* Now the main SPMD program, which simply finds whether this process
   is the root or not, and decides which function to call */

int main(int argc, char **argv)
{
  int myid, numprocs;
  MPI_Aint int_length;
  int array_of_blocklengths[2] = {1, 1};
  MPI_Datatype array_of_types[2];
  MPI_Aint array_of_displacements[2];

  /* Initialise MPI */

  MPI_Init(&argc,&argv);

  /* We tell MPI about the data packet we are going to want
     to send between processes */

  /* 1. Find the length of the relevant variables to determine offsets
     into structure. In this case just the length of the integer is
```

```
46     important. Awkwardly we can't define an array in C once we have
started
47     performing processing, and can't find the length of an MPI_INT
48     without processing....so we alter the array_of_displacement
49     element by hand ! */
50
51  MPI_Type_extent(MPI_INT, &int_length);
52  array_of_displacements[0] = 0;
53  array_of_displacements[1] = int_length;
54
55  /* 2. Define the array of types which go into this structure*/
56  array_of_types[0] = MPI_INT;
57  array_of_types[1] = MPI_FLOAT;
58
59  /* 3. Tell MPI about the structure */
60
61  MPI_Type_struct(2,
62           array_of_blocklengths,
63           array_of_displacements,
64           array_of_types,
65           &workDataPacketType);
66
67  /* 4. Commit the structure to MPI's internal memory */
68
69  MPI_Type_commit(&workDataPacketType);
70
71  /* The workDataPacketType is now something MPI knows about */
```

Now we have derived our application specific datatype, we can use workDataPacket-Type wherever we might use MPI_INT etc in MPI.

```
72  /* Find out the rank of this process */
73
74  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
75
76  if (myid == 0)
77  {
78    /* I am the root process */
79    master_process();
80  }
81  else
82  {
83    /* I am only a worker */
84    worker_process();
85  }
86
87  /* and shut down the MPI communicator */
88
89  MPI_Finalize();
90}
91
92/* ***************** MASTER PROCESS ****************** */
```

If we are executing on the 0th processor (though the meaning of this may vary from implementation to implementation) we perform the master process. This splits up the work to be performed, and sends it to the workers.

```
93void master_process()
94{
95  int num_workers;
96  int mytid,info;
97  float cur_value,area;
```

```
98  float val_increment;
99  int num_strips_for_this, average_num_strips, num_strips_remaining;
100  int cur_worker;
101  work_packet cur_work;
102  MPI_Status status;
103
104  /* We assume the workers are now waiting to be told
105     what to do */
106  MPI_Comm_size(MPI_COMM_WORLD,&num_workers);
107
108  /* The num_workers we receive includes the 0th, i.e., the root,
109     so we decrement so we can deal with this */
110  num_workers--;
111
112  /* We send them the start value, and the number of strips (of fixed
113     width) we want the worker to process. Note we use fixed width
114     strips so we can guarantee the same answer should be arrived at
115     no matter how many workers we use. This means we will always
arrive
116     at an approximate answer. In a real application we would not
wish to
117     do this. */
118
119  num_strips_remaining = 100;
120  average_num_strips = (int) (num_strips_remaining / num_workers);
121  cur_value = -4.0;
122
123  printf("Trying to send an average of %d strips to each worker\n",
124      average_num_strips);
125  fflush(stdout);
126
127  for(cur_worker=1; cur_worker<=num_workers; cur_worker++)
128  {
129    /* First calculate the number of strips which will be sent to
130     the current processor. Note that if an odd number of processors
131     are specified the last processor gets slightly more work */
132
133    if(num_strips_remaining < (2 * average_num_strips))
134    {
135      /* We are on the last processor, so we assign all remaining
136      strips to it, which may be slightly above the average */
137      num_strips_for_this = num_strips_remaining;
138      num_strips_remaining = 0;
139    }
140    else
141    {
142      num_strips_for_this = average_num_strips;
143      num_strips_remaining -= average_num_strips;
144    }
145
146   /* Now send a message to the worker processor telling it that it
must
147       process num_strips_for_this, starting at cur_value */
148
149    /* INSERT YOUR CODE HERE */
150    cur_work.num_strips = num_strips_for_this;
151    cur_work.start_value = cur_value;
152
153   printf("Sending %d strips starting at %f\n", cur_work.num_strips,
154        cur_work.start_value);
155    fflush(stdout);
156
157    MPI_Ssend(&cur_work, 1, workDataPacketType,
```

```
158           cur_worker,
159           1, MPI_COMM_WORLD);
160
161    /* END INSERT */
```

Nw we have derived our datatype the send operation is extremely simple. We simply fill the relevant data into the structure, and send the structure. The MPI implementation will perform any conversion that may need to occur internally.

```
162    /* After sending the message we increment cur_value for the next
163       iteration */
164    cur_value += (0.08 * num_strips_for_this);
165  }
166
167  /* Now we read in the results from each worker*/
168
169  area=0.0;
170
171  for(cur_worker=1; cur_worker<=num_workers; cur_worker++)
172  {
173    /* Read data from worker */
174    /* INSERT YOUR CODE HERE */
175    float tmp_area = 0.0;
176
177    MPI_Recv(&tmp_area, 1, MPI_FLOAT, cur_worker, MPI_ANY_TAG,
178          MPI_COMM_WORLD, &status);
179
180    area += tmp_area;
181
182    printf("Received value of %f from worker %d\n",
183        tmp_area,
184        cur_worker);
185    /* END INSERT */
```

The receive operation consists of blocking until we have the relevant area from a particular process.

```
186  }
187
188  printf("Area under function = %f\n", area);
189
190}
191
192/* ***************** WORKER PROCESS ***************** */
```

If we aren't executing as the master, then we perform this function, which simply receives the area & number of strips. Performs the calculation and then returns the result.

```
193void worker_process()
194{
195  float current;
196  float so_far = 0.0;
197  int mytid, parent, cur_strip;
198  int total_number_of_strips;
199  float start_value;
200  work_packet cur_packet;
201  MPI_Status status;
202
203  /* Read a message describing the start value, and number of strips
204     from the parent */
```

```
205
206  /* INSERT YOUR CODE HERE */
207
208  MPI_Recv(&cur_packet, 1, workDataPacketType, 0, MPI_ANY_TAG,
209        MPI_COMM_WORLD, &status);
210
211  /* END INSERT */
```

The receive operation is straightforward. Note that we use MPI_ANY_TAG merely to indicate that it could be used here, we could be more strict.

```
212  /* Then process that area of the function, accumulating the result
in
213     so_far */
214  current = cur_packet.start_value;
215
216  for(cur_strip=0; cur_strip<cur_packet.num_strips; cur_strip++)
217  {
218    so_far += integrate_strip(current,0.08);
219    current += 0.08;
220  }
221
222  /* Now send a message back to the master, containing the so_far
value */
223
224  /* INSERT YOUR CODE HERE */
225
226  MPI_Ssend(&so_far, 1, MPI_FLOAT, 0, 1, MPI_COMM_WORLD);
227
228  /* END INSERT */
```

Having performed the calculation we executing a synchronous send to the root (operating on processor 0).

```
229}
230
231float function(float value)
232{
233  /* Defines the function x squared plus 2 */
234  return(value*value + 2);
235}
236
237float integrate_strip(float start_value, float width)
238{
239  /* This function approximates the area under 'function' by using
240     a simple Newton-Raphson approach */
241
242  float tmp;
243
244  tmp = (function(start_value) + function(start_value+width))/2.0;
245  return(tmp * width);
246}
247
```

A.g Fortran Version

This is the Fortran version of the second exercise.

```
1 c This is the sample answer for the second exercise in the MANTEC
2 c "Introduction to MPI" course.
3
4       program main
5
6       include 'mpif.h'
7
8       integer myid, rc
9
10       integer aob(2)
11       integer aod(2)
12       integer aot(2)
13       integer wPType
14       integer int_extent
15       integer address
16
17c Define the datatypes that we will use to pass around
18c Use a common block to make the data contiguous in memory ?
19
20       integer num_strips
21       real start_value
22
23       common / workPacket / num_strips, start_value
24
25c First initialise the MPI system, which sets up the communicator
26
27       call MPI_INIT( ierr )
28
29c Now define the derived datatypes for MPI's benefit.
30
31       aob(1) = 1
32       aob(2) = 1
33
34       aod(1) = 0
35       call MPI_TYPE_EXTENT(MPI_INTEGER, int_extent, ierror)
36       aod(2) = int_extent
37       aot(1) = MPI_INTEGER
38       aot(2) = MPI_REAL
39
40c Now construct the Fortran equivalent of the structure in MPI-speak
41
42       call MPI_TYPE_STRUCT(2, aob, aod, aot, wPType, ierror)
43
44c Now tell MPI about it!
45
46       call MPI_TYPE_COMMIT(wPType, ierror)
47
48c Now obtain the rank of ~this~ process
49
50       call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr)
51
```

We have now defined the datatype we will use later on.

```
52c Now determine what to do
53
54       if(myid .eq. 0) then
55          call master_process(wPType)
56       endif
```

```
57
58      if(myid .gt. 0) then
59         call worker_process(wPType)
60      endif
61
62c Shut down the MPI communicator
63
64      call MPI_FINALIZE(rc)
65
66      end
67
```

First define the component functions that we will use.

```
68c     That was the end of the main program, now define the functions
69c     used by it
70
71      subroutine integrate_strip(start_value, width, result)
72      real start_value, width, result, tmp1, tmp2
73      real summation
74
75c     Do simple Newton-Raphson approximation
76
77      call myfunction(start_value, tmp1)
78      call myfunction(start_value + width, tmp2)
79
80      summation = (tmp1+tmp2)/ 2.0
81      result = summation * width
82      end
83
84      subroutine myfunction(value, result)
85      real value, result
86
87      result = (value * value) +2.0
88      end
89
90c ***************************************************************
91c Now the main functions, which are called by the SPMD main
92c program.
93c ***************************************************************
94
95      subroutine master_process(mytype)
96
97      include 'mpif.h'
98
99c This is the master process which doles out the work to the workers
100      integer nw, mytype
101      integer num_this
102      integer average_num_strips
103      integer num_remaining
104      integer curw
105      integer status(MPI_STATUS_SIZE)
106      real cur_value, area, tmp_area
107      integer myid
108
109      integer num_strips
110      real start_value
111
112      common / workPacket / num_strips, start_value
```

Define a common block so that num_strips & start_value match the derived datatype pattern.

```
113c Find out how many processes the user has created for us
114
115      call MPI_COMM_SIZE( MPI_COMM_WORLD, nw , ierr)
116      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr)
117
118c The number we receive includes the the root, so decrement
119
120      nw = nw-1
121
122      num_remaining = 100
123      average_num_strips = INT(num_remaining / nw)
124      cur_value = -4.0
125
126      print *, "Trying average of ", average_num_strips, " to ", nw
127
128c Now go into loop doling work out
129c Start after the root
130
131      curw = 1
132
133 10   if(curw .le. nw) then
134
135c First calculate the number of strips which will be sent to the
current
136c processor.
137         if(num_remaining .lt. (2 * average_num_strips)) then
138            num_this = num_remaining
139            num_remaining = 0
140         else
141            num_this = average_num_strips
142            num_remaining = num_remaining - average_num_strips
143         endif
144
145c Now send a message to the worker process telling it to process
146c num_this strips, starting at cur_value
147
148c ******INSERT YOUR CODE HERE******************
149         num_strips = num_this
150         start_value = cur_value
151
152         call MPI_SSEND(num_strips,1,mytype,curw,2,
153     $        MPI_COMM_WORLD,ierror)
154
155      print *, "Sending ",num_strips, " starting at ", start_value
156c ********END INSERT *************************
```

Fill the relevant data into the variables in the common block, then send this as the derived datatype mytype.

```
157         cur_value = cur_value + (0.08 * num_this)
158         curw = curw+1
159c Now loop back to work out how many we send to the next processor
160
161         goto 10
162      endif
163
164c Now we can read all the results back in from the workers
165
166      area = 0.0
167      curw = 1
168 20   if(curw .le. nw) then
```

```
169
170c Read data from workers
171c ***********INSERT YOUR CODE HERE ***************
172         tmp_area = 0.0
173
174         call
MPI_RECV(tmp_area,1,MPI_REAL,curw,1,MPI_COMM_WORLD,status,
175     $        ierr)
176         area = area + tmp_area
177
178         print *,"Received ",tmp_area," from ",curw
179c *************END INSERT************************
```

Receive a single real number from each worker in turn.

```
180         curw = curw+1
181         goto 20
182      endif
183
184      print *, "Area under function = ", area
185      end
186
187      subroutine worker_process(mytype)
188      include 'mpif.h'
189
190c This is the worker process
191      real current, so_far
192      integer cur_strip
193      integer status(MPI_STATUS_SIZE),mytype
194      real tmp
195      integer test
196      integer myid
197
198      common / workPacket / num_strips, start_value
199
200      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr)
201
202c Read a message describing the start value, and number of strips
from
203c the parent
204c ************INSERT YOUR CODE HERE****************
205
206      call MPI_RECV(num_strips,1,mytype,0,2,MPI_COMM_WORLD,
207     $     status,
208     $     ierror)
209
210
211c *************END INSERT *************************
```

Receive the derived datatype into the common block we define above.

```
212c Process that area of the function, accumulating result in so_far
213
214
215
216      current = start_value
217      cur_strip = 0
218      so_far = 0.0
219
220 40   if(cur_strip .lt. num_strips) then
221         call integrate_strip(current, 0.08, tmp)
222         so_far = so_far + tmp
```

```
223         cur_strip = cur_strip +1
224         current = current + 0.08
225
226         goto 40
227      endif
228
229c Now send a message back to the master containing the so_far value
230c *************INSERT YOUR CODE HERE ************
231
232      call MPI_SSEND(so_far,1,MPI_REAL,0,1,MPI_COMM_WORLD,ierr)
233
234c **************END INSERT **********************
235
236      end
237
```

Finally send the so_far variable to the master as a real number.

B Resource List

B.a Introduction

The standardisation process for MPI has been underway for sometime, but as mentioned in the course, most implementations at present are proof-of-concept at present, and therefore leave something to be desired.

As MPI is relatively new there is little additional material available for particular implementations (in the form of extra libraries and example applications for example) but hopefully this should change soon.

B.b MPI Implementations

The main public domain implementations are:

- ANL/MSU Freely available portable MPI implementation (mpich) ; available from

  `info.mcs.anl.gov/pub/mpi`

- Mississippi State University UNIFY implementation (provides a subset of MPI within the PVM environment, without sacrificing the PVM calls already available) , which can be obtained from: `ftp.erc.msstate.edu/unify`
- Edinburgh Parallel Computing Centre CHIMP implementation. This may be obtained from `ftp.epcc.ac.uk/pub/chimp/release`
- Ohio Supercomputer Center LAM implementation. Available from `http://www.osc.edu/lam.html`.
- MPI for the Fujitsu AP1000 from the Australian National University. From (note the ftp) `ftp://dcssoft.anu.edu.au/pub/www/dcs/cap/mpi/mpi.html`.

We have used mpich in the development of this course.

B.c Extra Resources

MPI is being actively used by many research groups around the world, and so it is likely that additional material wil become available soon. The best place to keep track of MPI material is on the WWW at

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

B.d Documentation

The principal documentation for MPI is the users guide which comes as part of each distribution. However there is one published book which is devoted to MPI which has recently become available:

"*Using MPI*: *Portable Parallel Programming with the Message Passing Interface*", William Gropp, Ewing Lusk and Anthony Skjellum

The book is available from:

The MIT Press
Book Order Department
55 Hayward Street
Cambridge, MA 02142.

Many parallel programming texts cover MPI to some extent, though no other single book really concentrates on this library. A good recent book which falls into this category though is

"*Designing and Building Parallel Programs*", Ian Foster, Addison-Wesley,1995.

An on-line version of this book is available on WWW at

`http://www.mcs.anl.gov/dbpp`