Introduction to Parallel Computing with MPI

Day 2: The MPI Standard

Course Introduction

Course Outline

- ❑ Background to parallel programming.
- ❑ Programming paradigms.
- ❑ The background to MPI, why and how?
- ❑ Message Passing according to MPI
- ❑ Programming with MPI
- ❑ Exercise
- ❑ SPMD & MPI
- ❑ Message passing
- ❑ Derived Datatypes
- ❑ Exercise
- ❑ Summary

Timing

10:00 - 10:15	Background to Message Passing
10:15 - 10:45	Background of MPI
10:45 - 11:15	Programming with MPI
11:15 - 12:00	First exercise
12:00 - 1:00	*Lunch*
1:00 - 1:45	Basic Synchronous M.P.
1:45 - 2:30	Asynchronous MP in MPI
2:30 - 3:15	Deriving New Datatypes
3:15 - 4:00	Exercise 2

Introduction to Message Passing

Parallel Programming

- ❑ Why Parallel Programming?
- ❑ Certain classes of problems are either:
  - Too large for available serial architectures.
  - Take too long to execute on serial machines.
  - Don't handle large loads well, e.g., a database server which operates well with 10 users, but when 100 people use it the performance suffers.

The promise of Parallel programs

- ❑ So, as a programmer, we are forced to turn to parallel machines as a possible solution to our problems.
- ❑ Parallel processing claims to be
  - Cheaper, in terms of Price/Performance.
  - Faster than equivalently expensive uniprocessor machines.
  - Scalable : the performance of a particular program may be improved by execution on a large machine.
  - Reliable? In theory if processors fail we can simply use others.
  - Handle bigger problems.
- ❑ But to use parallel machine we must employ *parallel programming*.

What is Parallel Programming?

- ❑ Concurrent operation of elements within a system
- ❑ Low-level hardware has been parallel for many years. e.g., overlapped I/O, multi-tasking. Traditionally this parallelism is hidden from the user.
- ❑ Machines which allow the user to take advantage of parallelism are normally referred to as *parallel machines*, to distinguish from conventional architectures.

How do we take advantage of parallel machines?

- We must identify potential parallelism in *applications*.
- If we can find sections of computer code which can be executed at the same time as other sections, *without changing the results generated by that program*, then we have found potential parallelism.
- Before we can discuss how we exploit this parallelism we review the machines.

Parallel Machines

- A wide variety of parallel architectures exist.
- Fortunately there exists a useful taxonomy which we can employ to categorise them (Flynns taxonomy).
- This taxonomy categorises machines dependent on how each handles instructions (multiple programs), and data.

Flynns Taxonomy

- The taxonomy breaks into :
  - SISD - Single Instruction/ Single Data. This category corresponds to conventional serial architectures.
  - MISD - Multiple Instruction/ Single Data. Here the machine lets several processors execute different instructions on *one* data stream in a pipeline fashion.
  - SIMD - Single Instruction/ Multiple Data. Here a single program is executed on several pieces of data simultaneously.
  - MIMD - Multiple Executions being executed on separate data simultaneously.
- This abstract taxonomy provides a general indication of the capabilities and programming style required for a particular machine, but more information is usually needed.

Other Considerations

- Grain Size : The size of processes which we execute on processors affect how we can use the machine.
- Interconnection : How processors communicate with one another.
- Coupling : How processors communicate with memory.
- Programming Mechanism. Whilst a wide variety of abstract parallel programming styles exist (one of which is the principal topic of this course), not all are useful on all machines.

Parallel Programming

- As computer science has yet to develop compilers which automatically use parallel machines well, we must program in special ways.
- Various paradigms exist, but we are interested in the *message passing* technique.

Message Passing

- For message passing to be a viable means of exploiting parallelism we conventionally employ it on MIMD machines.
- The *application* is split into a number of *programs*. Each program operates ‘independently’, usually on different processors.
- The logic of the application is maintained by coordinating the component programs through the exchange of messages.
- The maintenance of this underlying logic, which controls how the application works is the responsibility of the *programmer*, not the machine.
- This makes this form of programming hard!

An Example

- Computer animation:

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

- Both steps are time consuming, they could be organised as two seperate programs:

Animation process

```
for(time=start; ...) {
    process_all_bodies;
    send_graphics_data()
}
```

Graphics process

```
for(...){
    receive_data();
display_graphics();
}
```


Facilities of MP Libraries

- Application programmers don’t want to deal with the messy aspects of getting processors to communicate.
- So they use message passing libraries which provide:
    - the ability to create processes on remote machines
    - the ability to monitor the state of these remote processes (More on these two points later in relation to MPI)
    - routines which enable messages to be sent reliably from program to program, without the programmer needing to know *how* this is achieved.

MP Implementations

- There are a large range of message passing libraries in use today, on a wide range of architectures.
- A programmer must choose between them, though they all perform similar functions, and so porting is not very difficult.
- We will concentrate on a recent message passing standard, for which several implementations are appearing.
- The Message Passing Interface MPI.

Introduction to the Message Passing Interface (MPI)

What is MPI?

- A proposed standard message passing interface to libraries.
- Provides explicit message passing for distributed memory machines and networks of workstations.
- Developed over two years by an international consortium to address the problem of having multiple competing libraries, all of which performed the same task, but used different approaches.

Rationale

- The message passing paradigm is widely understood, if not widely used *well*.
- Unfortunately each vendor has their own implementation of libraries, which is optimised for the architecture of their machine, and so isn't very portable.
- The MPI committee felt that this was hampering the adoption of message passing in the wider world.
- They identified the need for a generic portable library which defines a set of routines which can be efficiently implemented on a wide range of machines, and which together provide sufficient functionality.

MPI

- The result of their efforts, MPI, is intended to form a widely used standard, gradually replacing vendor specific interfaces, and other public domain libraries, such as PVM and p4.
- Vendors, once freed of the burden of developing their own library designs, may concentrate on implementing the library well for their architecture.
- However MPI was not sanctioned by any official standards body, which has the advantage that it may be modified far more quickly, but it means MPI is usually not a government requirement.
- Most major p. computer developers participated.

The MPI Initiative

- Began in April 1992
- Met every six weeks during ‘93.
- Group formed from over 40 Universities, commercial vendors and users.
- Meetings were open to everyone though.
- Significant commercial support from
  - Convex, Cray, Meiko, Intel, Thinking Machines, IBM, nCube, NAG, Parasoft, Shell & Arco
- European involvement was funded by Esprit.

Cont.

- Rather than start from scratch the MPI group sought to adopt the best features from existing implementations, including:
  - Intels NX/2
  - Express
  - PVM
  - p4
  - CHIMP (From Edinburgh)
  - Work at IBM TJ Watson Research centre
  - PICL
- This means that existing message passing developers will probably find something familiar in MPI!

What is in MPI

- The committee specified in the standard:
  - Point to Point communication
  - Collective communication routines
  - Support for grouping operations, i.e., ways of telling MPI to use an addressing scheme which makes sense to your application.
  - Mechanisms to separate communications in the same program (to enable libraries to be easily developed).
  - Bindings for C & F77
  - A profiling interface, as an aid to developers.
- Note that MPI specifies an *interface*, not how is internally implemented.

What ISN'T in MPI!

- Due to time constraints the committee decided to leave some message passing concepts out of the first MPI standard, planning instead to place them in the forthcoming MPI2. The more significant elements include:
  - Explicit shared memory operations, i.e., ways in which your application can take advantage of operation on certain architectures.
  - Any support for process management!, much more on this later!
  - Support for threads, i.e, the ability to have multiple threads of execution operating while sharing variables without explicit message passing (such as found on the KSR)
  - Debugging facilities
  - Parallel I/O

Summary

- Having outlined the background for the development of MPI we can look at the standard.
- An important point to remember at this stage is that MPI is a *piece of paper*, not a library!
- You will always be working with *implementations of a library* which conform to MPI.
- This pedantic distinction will become more important as you begin to work with such libraries!

Fundamentals of MPI

MPI Implementations

- Before we go on to discuss the fundamentals of MPI we must first outline the different implementations of MPI available.
- Each was originally written as a proof-of concept library, which enabled researchers to check the MPI standard for any inadequacies.
- This means that many of them are poorly documented, and some of them are unreliable, and not supported.
- Unfortunately until parallel computer manufacturers ship their own implementations of MPI libraries these are all we have to work with.

Cont.

- However, provided you use common machines, they provide a useful way of developing code which is 'future-proof'.
- The 3 main implementations at the time of writing are:
  - Argonne National Labs & MSU library : `mpich`
  - Edinburgh's CHIMP Implementation
  - Ohio Supercomputers LAM implementation
- Each of these implementations is built on top of an existing message passing library, for example `mpich` sits on top of either p4 or PVM.
- This may cause performance problems, but has enabled these implementations to be quickly developed.

MPICH

- For the purposes of this course we will use the ANL/MSU implementation of `mpich`, sitting on top of the p4 system.
- This is a popular implementation and was chosen because it operates on available hardware.
- In theory any implementation could be used.

The MPI View of processes

- We have already seen that MPI doesn't specify ways of managing processors on either a parallel computer or network of workstations.
- This is in stark contrast to other libraries, such as PVM.
- PVM programmers will be used to the system of writing a program which spawns other programs which execute on a virtual machine.
- *An MPI program cannot spawn other processes*, this has to be handled by other software which is implementation specific.

Starting processes

- In `mpich`, like most current MPI implementations, the user (not the programmer) specifies how many processes they want, and where, at *start up time*.
- This means MPI does not provide any means of dynamic parallelism, the user must know in advance how many processes they wish to use.
- This is a limitation which may be addressed in MPI2.

The SPMD Model

- Recall our simple animation example:

Animation process

```
for(time=start; ...) {
    process_all_bodies;
    send_graphics_data()
}
```

Graphics process

```
for(...){
    receive_data();
display_graphics();
}
```

- We normally think of this as two completely separate programs.
- However, as MPI doesn't specify how the user might cause two different programs to be executed, it wouldn't be very portable.

Cont.

- Consequently it is more normal for MPI programs to use a style of parallelism called SPMD, which means Single Program Multiple Data.
- This means you write a single program, which incorporates the functionality of both the logical programs.
- A program suitable for such use must detect at run time which functionality it must provide i.e., in this case whether its an animation or graphics process, and then do it.

An SPMD Outline

- In practice this means you will always write programs whose main routine looks like:

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

- In larger examples we may wish to have a more elaborate outline, but fundamentally the program must always decide what it is going to do at run time.
- Note that this isn't strictly a restriction imposed by MPI, but as the standard doesn't specify anything else it tends to be a portable approach.

MPI Preliminaries

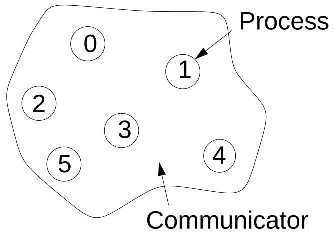
- A users application makes use of MPI functionality by calling functions provided by the implementation.
- Each MPI routine returns an error code, which in C is usually an `int`, and Fortran routines have an extra parameter so they can determine the functions success.
- All MPI functions in both languages start with the prefix `MPI_`.
- Fortran routines are all upper case, while the C bindings are mixed case.

Cont.

- To conform to the standard the library hides the internal details of its operation, and any internal details the application needs to know about are referred to by *handles*.
- In Fortran handles are of type `INTEGER`.
- In C each handle is separately `typedef`'d.
- In Fortran arrays are assumed to index from 1, in C from 0.

Communicators

- The focus of any message passing library is the ability to transmit data between programs.
- In MPI such messages can only travel within a communicator.
- A communicator is a set of processes which are assumed to know about one another.



Cont.

- Whenever an MPI message transmission function is called the program must indicate the communicator it wishes the message to pass through.
- A process may be a member of more than one communicator.
- The program must also indicate which process in the communicator the message is intended for (assuming point to point communication for now).
- This number is known as the *rank*.
- The rank is only relevant to one communicator, so if one process is in more than one communicator it may have a different rank in each.

MPI_COMM_WORLD

- When the program calls MPI_INIT the library will set up an initial default communicator, which *all* processes are a member of.
- This communicator is called `MPI_COMM_WORLD`.
- Why have the complexity of a communicator, what is wrong with a single process specific address for messages (like `tid` in PVM)?

Library Construction

- We already know that message passing application construction is painful!
- The best of making this easier is to employ libraries which do the basics for you (by which we mean higher level than MPI).
- Unfortunately for a library to be useful it must hide its implementation from you, so you just need to know *what*, not how, it does what you need.
- In single-tier message addressing it is very hard to hide messages from the application.
- For example if the library is sending messages how do you ensure the application doesn't pick up messages not meant for it?

Cont.

- With MPI this can easily be achieved by using different communicators for different libraries, as messages from one communicator do not interfere with other comms.
- We have now covered the fundamentals of the MPI standard design, and we can begin to see how you can write programs with it.

Programming in MPI

Starting MPI

- The first MPI routine called in any program must be MPI_INIT (this must only be called once).
- The C version of this takes command line parameters:

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

- The Fortran version takes only the error code.

```
MPI_INIT(IERROR)
    INTEGER IERROR
```

- This call allows the MPI implementation to perform any necessary initialisation, though obviously what is required will vary from implementation to implementation.

Shutting down MPI

- After your program has finished all its work it must tell the MPI implementation that it can shut itself down.
- This allows MPI to cancel any outstanding messages, free any memory allocated for its use and so on.
- After this call no more MPI functions may be used (including MPI_INIT).

```
MPI_FINALIZE()
```

- Alternatively if one process wishes to force all processes in a specific communicator to stop then it calls

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

- If `MPI_COMM_WORLD` is used then the whole program will stop.

A Basic Program

- Like any other library the application must include a header file which describes the functions and variables which may be used.
- In C this requires

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

- and in Fortran we use

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


Accessing Comm. Info

- As we have mentioned knowing the rank of processes is critical for us to perform any message passing.
- MPI provides two functions which are pertinent to this:

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

- which returns the rank of the calling process in the named communicator, and,

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

- which returns in size the number of processes in the named comm.

Exercise 1

- We have now explained
  - How MPI views addressing issues.
  - The sort of programming model that MPI *encourages*.
  - How an MPI implementation is used by the application.
- To reinforce this we now set a very simple exercise:

Write a program which uses MPI to start 2 child processes, each of which tries to print a hello message to the screen, and then exits.

MPI on the HPs

- ❑ Before you can proceed with this exercise there are a couple of implementation specific pieces of information you will need.
- ❑ Whilst these may not apply to you when you come to use MPI for real work, because you will always be using an implementation of the spec., there will always be something you need to know.
- ❑ For the purposes of this course we are going to use `mpich` on a network of HP workstations, which all use AFS.

Spawning Processes

- ❑ For a network of workstations to be useful for parallel processing it must be possible to sit at one, and cause processes to execute on others.
- ❑ Unfortunately AFS (and Kerberos) gets in the way of the most common technique (which is to use UNIX's `rsh` command)
- ❑ Therefore we must ensure that a server process is executing on each machine in the network we are going to use.
- ❑ This server process listens to incoming requests and messages on a UNIX port.
- ❑ The server, which is called `serv_p4`, must be executed by the user who will employ it.

Cont.

- ❑ To use these the user must have two UNIX environment variables set, which indicate to the MPI library how it goes about using the library.
- ❑ These have already been set up for the student accounts, but if you wish to continue to use MPI on the HP's you may need to know about them:

```
setenv MPI_USEP4SSPORT yes
setenv MPI_P4SSPORT <number of UNIX port>
```


Running your program

- ❑ Once you have compiled your program you execute it by using the mpirun command.

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

- ❑ Note that we specify the number of processors, and on each an instance of `<progname>` is executed.
- ❑ This constructs a file which indicates what programs are run where, and then proceeds to execute them.
- ❑ This command finds the list of machines from a file within the MPI installation (out of your control)

Makefile

- A makefile has been placed in the relevant subdirectories of each of the guest accounts.
- For this exercise if you intend to write in C you should call your file `ex1.c`
- If you are more comfortable in Fortran write a program called `fex1.c`

Exercise 1

- To reiterate the first exercise requires you to :

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

- We provide a makefile in the $HOME/ex1 directory.
- Some reminders:
  - You will need to write one single (SPMD) program.
  - Each program needs to include “mpi.h”, or “mpif.h” in order to prototype the various MPI functions.

Message Passing in MPI

More advanced MPI

- In order for MPI to be any use to use in our production of parallel applications we must understand how the message passing facilities work.
- In MPI a message is an array of elements each of a particular MPI datatype

Integer	Integer	Integer	Integer	Integer

- The messages are typed in the sense that the contents’ type must be specified in both send and receives.
- So when the programmer comes to make a send call, they must, at *call-time*, specify the contents of the message.
- Before discussing how we first look at what types MPI knows about.

MPI Types

- The basic C data types in MPI are

MPI Datatype	C Datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	singled long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

- Note that the C datatypes do not always have to be specified in full (i.e., by default ints are signed).

Cont.

- Fortran types are

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

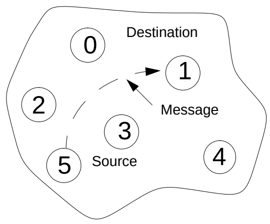
- Note that
  - There is some overlap.
  - MPI implementations are assumed to perform any necessary conversion between different architectures transparently....as far as the application is concerned a float is a float is a float!

Constructing Messages

- As mentioned in MPI messages are assumed to be a sequence of datatypes.
- However, a message can only contain a sequence of data items, *all of which are the same type*.
- This is particularly useful if we wish to send around arrays of data, but for anything more flexible it seems horribly restrictive.
- MPI provides facilities which allow the application to describe new datatypes, and in doing so allows sufficient flexibility to perform useful work.
- Having looked at messages we can examine the forms of communication allowed by MPI.

Point to Point Comms.

- There are several styles of communication available in message passing, but for now we will concentrate on point to point.
- In other words a single process sending a message to another single process.



- We have already seen that messages travel between ranked processes in a common communicator.

Communication Modes

- For the construction of satisfactory parallel applications it is necessary to have greater control over message transmission than simply "It goes there"!
- MPI provides 4 communication *modes* which the application specifies:
  - Synchronous send
  - Buffered Send
  - Standard Send
  - Ready Send
- We will discuss each of these in turn, and attempt to identify why you might want to use each type in turn.

Synchronous Send

- Quite often in the construction of distributed applications we need to know that the destination process has received the message *and is acting on it* before proceeding.
- For example we may need to keep a record of which process is working on which unit of work, so we can know which we are waiting for.
- We cannot do this easily without being sure that the relevant processor received the messages it was sent, and is working on them.
- Before we discuss how this is achieved in MPI its worth reviewing the basics of message passing.

Data Transmission

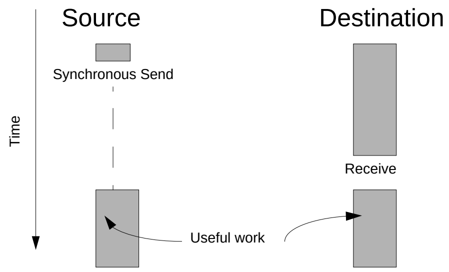
- A large variety of networking technologies exist, all of which are particularly well suited to certain situations.
- Whilst message passing libraries (like PVM and MPI) hide this complexity, we still need to be aware of some of the details of the networks.
- The most important point we need to understand is the buffering used by most networking systems.
- As most networks cannot guarantee a particular level of service (speed) they tend to let each computer place comms. data in buffers, and all communication actually goes to and from these buffers, not the application.

Cont.

- This means that communication takes the form of
  - Source application places the data in the buffer
  - Network software arranges for contents of buffer to be transferred to destination buffer.
  - Destination application copies data out of buffer.
- Notice that after the first step the source application has done its work. However it cannot know how long it will take for the third step to be reached.
  - The network may be congested and the data may take a long time to reach the destination.
  - The destination application may still be working on something else.

Back to S. Send.

- The MPI version of the synchronous send style hides the buffering, and won't let the source application continue work until the destination application has received the data *into the application*, not just its local buffer.



- Note that in extreme cases the synchronous send operation may cause the source to wait for a long time!

S. Send in MPI

- In MPI the synchronous send call looks like

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

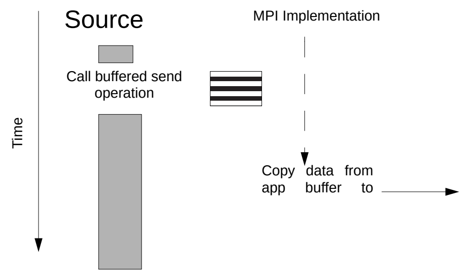
- Taking each parameter in turn:
  - The 'buf' is the address in the application where the data is currently residing.
  - The 'count' is the number of data items, of type 'datatype' (which is one of the MPI types we talked about earlier) which is in 'buf' that we want to send.
  - The 'dest' is the *rank* of the destination relative to the communicator 'comm'.
  - The 'tag' is an application specific identifier which we associate with the message. We'll go into more detail about this when we discuss receives.
- In this function MPI copies the data from `buf` into the comms buffers, so this increases the overhead.

Performance Issues

- As we mentioned synchronous sends can be very useful, *but* they can slow down our application.
- We would like to do useful work in the time it may take the message to reach the destination machine, and the destination application to receive the message.
- MPI (like all other messaging libraries) allows us to achieve this by buffered sends.
- Essentially these work by adding an extra buffer, which is within the application, which the communication library will load the data from when it comes around to sending the data across the network.

Buffered Send

- Essentially we can think of this as:



- Note that the underlying network software, which is operating as another task in the OS, will eventually copy the data from the app. buffer into its own internal buffer for eventualy transmission.
- Meanwhile the application can work on. (The completion is local)

In MPI

- For buffering to be useful in MPI the application programmer must manage the app. buffer him or herself.
- This is achieved by the programmer allocating space in the program (in 'C' using `malloc`, and in Fortran an array), which we then tell MPI about.
- We inform MPI about this space using

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

- This tells MPI to use the space of `size` bytes starting at address `buffer` as the app. buffer for subsequent buffering sends.

Cont.

- Once we have finished, or we wish to change the buffer in some way, we must tell MPI to stop using that buffer.

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

- Note that this doesn't deallocate the actual memory from the *applications use*.
- If you wish to do that use C's `free()`
- Once we have our buffer set up we call the buffering send function:

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

- Remember that MPI copies your data from `buf` to the current application buffer, not you!
- There is an overhead to this.

Other Comms Modes

- The two communication modes we've talked about so far are the most important, but MPI provides 2 more modes for specialist uses.
- You probably won't use these very much when you start programming in MPI, so we won't devote too much time to them.
- They are
  - Standard Mode
  - Ready Mode

Standard Send

- MPI provides another, less straightforward, communication mode, the *standard send*.
- The standard send completes once the message has been sent, which may or may not imply that the message has arrived at its destination.
- The message may 'lie in the network' for some time.
- Internally MPI implementations may use either buffered or synchronous sends.

Ready Sends

- The point to point communication modes we have discussed so far all ensure that eventually the message will be received.
- In *some* circumstances we can relax this restriction to “the message will be received if the destination can take it”.
- A ready send completes immediately. If the destination has already posted a receive (i.e., is waiting for incoming data) then it receives the message.
- If the destination isn’t waiting, then what happens *isn’t defined*.
- This ambiguous statement in the spec. can normally be taken to mean that the message may be dropped.

Cont.

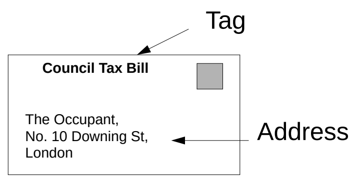
- But the sender has no way of knowing what happens.
- Obviously this strange form of communication is only useful in very restricted cases, and most MPI users will never need to use it.

Receiving in MPI

- We have now discussed, at some length, some of the ways in which an MPI program can send messages.
- We can now discuss the manner in which messages can be received.
- In MPI the most common form of a receive function is the blocking receive (we will discuss more advanced forms later).
- This form of receive is a function which the application calls, passing as parameters the type of message it expects to receive.
- But how does it know what type a message is before it receives it?

Tags

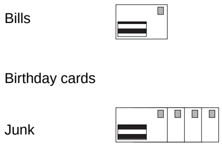
- When we first introduced sends we mentioned the concept of applying a tag to a send call.
- A tag is a piece of information associated with a message, but which isn’t contained *within* it.



- The most apt real-world example of this being lettering placed on an envelope which you can read before opening.
- This is useful if you wish to ignore bills!

Cont.

- In much the same way real-world labels help you classify post, tags in MPI messages enable your application to choose which messages it wishes to deal with.
- In MPI, like most message passing libraries, messages arriving at a particular host are buffered (yet another buffer!).
- These messages are normally arranged in queues (the ordering being based on the time of receipt), with different queues for different tags.



Receives

- In MPI when we wish to receive messages the receive function looks at these queues, and decides which one to return to the application.
- If we specify a particular tag then MPI will return the first from that queue (or wait until a message of that tag arrives).
- If we aren't specific about the tag type then MPI will examine each of the queues, find the message which arrived first, and return that.
- We indicate our preferences by a parameter to the receive function.
- Tags in MPI are Integers, so the application must choose meaninful numbers.

Cont.

- The blocking receive function in MPI is

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

- The parameters being:
  - Return a message of tag 'tag', which came from the 'source' (which is specified as a rank in the communicator 'comm').
  - Place the message, which consists of 'count' instances of type 'datatype', in 'buf'.
- The tag and source params. can be
  - Source is either a rank, or MPI_ANY_SOURCE
  - Tag is either a meaningful number of MPI_ANY_TAG
- The 'status' parameter is of type MPI_Status, which we will now discuss.

MPI_Status

- If an application decides to use a wildcard in either the source or tag then it may receive a variety of different messages.
- However we may still need to know either what the message is, or who it is from, before looking at the buffer.
- MPI allows this by setting a structure (the status) with these pieces of information.
- In C this is a structure, which contains:

```
status.MPI_SOURCE /* The rank of the source */
status.MPI_TAG /* The Tag of the message */
```

- In Fortran we query this using two functions:

```
STATUS(MPI_SOURCE)
STATUS(MPI_TAG)
```

An Example

- ❑ An extract from an MPI program sending a single float to the process rank 0 in MPI_COMM_WORLD would be:

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

- ❑ An extract from the rank 0 process:

```
float result;
int result_from;
MPI_Status status;

MPI_Recv(&result, 1, MPI_FLOAT, MPI_ANY_SOURCE,
        MPI_ANY_TAG,
        MPI_COMM_WORLD, &status);

result_from = status.MPI_SOURCE;
```

- ❑ Note that the receive doesn't care what tag or source, but can determine these later on using the status variable.

Cont.

- ❑ The wildcard source and tag parameters seem to provide us with great flexibility in constructing our parallel programs.
- ❑ However it is worth remembering that the receive operation for wildcards still copies the data into the buffer.
- ❑ If the buffer isn't large enough, or the wrong datatype is there, an error still occurs.
- ❑ Therefore wildcards must only be used where all the messages could fit into the buffer, and have exactly the same datatype.
- ❑ Obviously this is very restrictive!

Non-Blocking Comms.

Blocking vs. Non

- ❑ The modes synchronous, standard and ready sends are called *blocking* comms.
- ❑ This means that the program will only carry on executing when the comms. have been successful.
- ❑ As we discussed in the buffered mode this causes performance penalties. One way round this problem is to use buffered send.
- ❑ But
  - ■ The user must manage the buffers, and,
  - ■ it takes some time to copy the data into buffers.
- ❑ So MPI provides an additional suite of comms modes, which are *non-blocking*.

Theory

- A non-blocking communication in MPI (like most MP libraries) is a two step process.
- You call the send function, with the usual send parameters, which completes *immediately*.
- At some point in the future you may call another function which will tell you whether that data has been sent out..
- Essentially this second call tells you whether you can reuse the memory you were sending from.
- It does not always mean that the destination application is acting on that message.

Non-Blocks in MPI

- There are 3 special non-blocking sends in MPI, and a non-blocking receive.
- We deal first with the sends.
- Each of the 3 send types terminates immediately (form the calling programmers point of view) but the whole communication is only complete later.
- With synchronous sends it completes when the matching receive on the destination has *started*.
- The situation is slightly more confused if there is a non-blocking receive at the other end..more on this later.

NB Sync. Sends

- The non-blocking synchronous send call

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

- There are two items worthy of note:
  - The I in the title denotes non-blocking (immediate), and all non blocking comms have this.
  - There is an additional ‘request’ data item in the call.
- The ‘request’ data item is the ‘magic token’ that MPI attaches to this *particular message*.
- Each non-blocking send has a request associated with it, of type `MPI_Request`.
- Recall that the second stage of n.b. comms involves asking whether the comm. finished....request allows this.

Completion

- After we call a non-blocking communication we will need to find out whether it completed.
- To do this we call an MPI function. There are two functions, MPI_WAIT will block until the relevant message has completed

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

- and MPI_TEST will return immediately.

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

- Here flag is set to TRUE or FALSE.
- An example program might be:

```
MPI_request my_request;
MPI_Issend(....., &my_request)

/* Do some useful work */
/* Then wait for message to have been sent */
MPI_Wait(my_request, status);
```

N.B. Receives

- We have now looked at the basic forms of non-blocking sends. However MPI also provides non-blocking *receives*.
- Such operations are occasionally needed because receives can be time-consuming, particularly where large amounts of data must be moved, and some conversion is required.
- The receive is initiated by:

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

- Note that requests are also associated with receives, and the same query routines (WAIT & TEST) are used to determine success.

Cont.

- We have already mentioned the difference between a send call terminating (which should occur immediately) and terminating (which occurs when an MPI_Test determines success).
- This termination condition for synchronous sends and blocking receives is straightforward.
- But for ssends and irecvs the *send* terminates when the receive call has been made, *not* when it terminates.
- This means that the destination may not actually be dealing with the message, in fact the application may be working on something else.
- This may make a difference to you!

Other NB Info

- There are also non-blocking forms of standard and ready sends.
- However, buffered, synchronous and n.b. synchronous are likely to be of most use to the MPI programmer, so we won't cover the other send modes.
- MPI also provides more advanced completion test routines, which allow the programmer to determine the success of a sequence of sends and receives.
- These operate by being provided with arrays of requests, which the routines test, and set arrays of successes.

Completion Tests.

- The additional completion tests are:

Test	Wait type	Test type
At least one, return exactly one	MPI_WAITANY	MPI_TESTANY
Every one	MPI_WAITALL	MPI_TESTALL
At least one, return all which completed	MPI_WAITSOME	MPI_TESTSOME

- *ANY will return with information about the first item of interest, it will block until the first change.
- *ALL will either block until they have all succeeded, or return info about all.
- *SOME is similar to ANY, but instead of only dealing with the first, will return information about any that have completed.

Cont.

- We have now described the basic messaging infrastructure provided by MPI, including
  - The addressing scheme,
  - The different send modes
  - Receives
  - Non-blocking sends and receives
  - More advanced completion tests
- We can now go on to discuss higher level messaging ops which build on this foundation.

Derived Datatypes

Derived Datatypes

- We have now examined some of the ways in which arrays of certain datatypes can be sent between processes using MPI.
- Whilst this is useful we also need to be able to send more complicated sequences of data.
- To give a useful example, in our second exercise we construct a simple program which integrates under a curve using a Newton-Raphson approximation.
- Our application consists of a single ‘root’ process, which divides the work up, and a number of worker processes (though obviously this is all combined in a single SPMD program).

Cont.

- At some stage then the root process needs to send information to the workers telling them:
  - At what parameter value to begin integrating,and,
  - How many strips (of fixed width) to calculate (this is a *very* simple program!)
- These are essentially two numbers, which in C we could represent as a structure, and in Fortran as elements of arrays:

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

- We wish to send this ‘work packet’ to our workers. But in MPI we have only seen how to send messages of a single datatype.

Multiple Messages

- We could send two separate messages (one with an integer, and another with a float) to each worker.
- In our simple application this might be sufficient.
- But, as we have seen, messages arriving on MPI hosts are queued dependent on arrival time. If a message arrived between these two messages, the programmer would have to sort out the mess!
- We would like to bundle these two separate data items into a single, ‘application meaningful’, item.
- MPI allows this by derived datatypes

New Datatypes

- In MPI it is possible to define new datatypes, which are meaningful to our application, which we can then tell MPI about.
- From then on we can use this new type exactly as we might use MPI_FLOAT and the other types.
- So for example we might send an array containing 5 of our new types using:

```
MPI_Ssend(&my_array, 5, MYNEWTYPE, 0,
            1,MPI_COMM_WORLD);
```

- For this to be possible both source and recipient of the message must know about this new type (i.e., have defined it), so type definitions tend to take place in the portion of SPMD code common to all processes.

Cont.

- Whilst it is easy to see (potentially) how we might use this to transmit structures (in C), it is also possible to define more flexible datatypes.
- It is also possible to
  - Define a datatype to be an arbitrary region of an array. This is extremely useful for matrix operations.
  - Have an datatype consisting of arbitrary data, which aren’t necessarily obviously connected in the languages view (C or Fortran) but make sense for the application.
- In theory the concept of building new datatypes can clarify the message passing application, as it is easier to see what is being passed around.

Making a new type

- The datatype construction process consists of two stages, performed at *run-time*:
  - Construct the datatype in an MPI structure designed for such a process. Rather than filling in its contents yourself you achieve this by calling MPI functions.
  - Commit the datatype, i.e., tell MPI about the type.
- After this you may use the type as though it was an MPI intrinsic type.
- Finally you may
  - Free the datatype, i.e., tell MPI to forget about it.
- This is good practice, as it allows the MPI implementation to reuse memory.
- We will now discuss these stages in turn.

Constructing a Type

- As you may know data, which is meaningful to computer languages such as C or Fortran, is simply stored in memory by a sequence of bytes.
- For example in C it is quite common for a `float` to require 4 bytes of storage.
- However unless you know that, at location 0x4025 (for example) the following 4 bytes together constitute a `float`, you would just see a sequence of 0's and 1's.
- The intrinsic MPI datatypes (MPI_FLOAT) for example, tell MPI this information.
- When you construct a new datatype you are essentially providing this *very low level* of information, which allows MPI to access your data.

Cont.

- The most important thing to remember is that you are *providing machine specific byte oriented information, which allows MPI to interpret your datatype once it knows the start address*.
- Why does it have to be this contorted?
- MPI internally has to be prepared to work with
  - Optimised communication hardware which requires this sort of information, and
  - languages which have difficulty providing higher level concepts such as structures.
- This makes your job of defining datatypes rather more confusing, but provided you are aware that you are calling functions, which determine byte offsets (etc) you should be ok!

Type Maps

- A derived datatype is defined (conceptually) using a type map:

Basic type 0 (e.g., MPI_INT)	Displacement of type 0
Basic type 1 (eg MPI_FLOAT)	Displacement of type 1
..	..
Basic type n-1	Displacement of tyoe n-1

- When you come to call a send with this new type, this map will be used to find the individual items which constitute you map.
- MPI will be provided with a start address in such a call, to which it will apply the displacements which it has been told about.
- The type map functions as a sort of stencil over memory.

Cont.

- Rather than require you to understand how your intrinsic datatypes are held in memory, MPI provides functions to determine this information in a form suitable for inclusion in a derived datatype.
- Awkwardly such information is easy to find in C, but MPI insists you use its functions!
- This function is

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

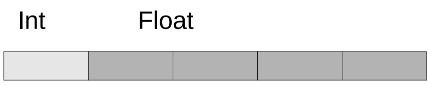
- This places the length of the defined 'datatype' in the 'extent' variable.

Structures

- Now we have discussed the fundamentals of deriving datatypes we can begin to talk about building different types. One of the more useful (for our exercise) is that of structures.
- The first thing we need to construct is our array of displacements for the structure we discussed earlier.

```
MPI_Aint array_of_displacements[2];
MPI_Type_extent(MPI_INT, &int_length);
array_of_displacements[0] = 0;
array_of_displacements[1] = int_length;
```

- Note that the first item is the integer, and the second the float, but we are looking for the offset, and therefore the offset for the float is the length of the int!



Types

- Once we have constructed our array of displacements the next thing we need to describe is the array of existing types.
- This should look like:

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

- You should notice that we can use previously defined datatypes here, so we could build new types out of the type we are currently defining!
- Once we have built this we need to build up one more array!

Blocks

- We have previously indicated that the entries in the arrays we have built up (displacement and types) correspond to entries in the *structure* (or memory).
- In fact this isn't quite true!
- MPI allows us to simplify our definition by letting us treat **chunks** of our structures as single items.
- Take the structure (which could equally be specified in Fortran):

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

- We might think this contains 12 different data items, so we have to fill in 12 entries in our arrays.

Cont.

- MPI lets us do this if we want, but also provides a short cut.
- As far as MPI is concerned a structure (in its way of thinking) is a sequence of *blocks*.
- Each block consists of 1-n datatypes, all of which must be of a single existing type.
- In other words a block is the sort of type we could send using a simple send function.
- Another way of thinking of that structure is
  - An integer
  - A block of 10 floats
  - An integer

Defining Blocks

- The entries in the displacement and type arrays correspond to blocks, not single datatypes.
- To tell MPI how many of the basic datatypes are in each block we define another array.

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

- For our exercise this is very simple.
- For the other structure this might look like:

```
int array_of_blocklengths[3] = {1,10,1};
array_of_displacements[0] = 0;
array_of_displacements[1] = int_length;
array_of_displacements[2] = int_length +
                  (10 * float_length);
array_of_types[0] = MPI_INT;
array_of_types[1] = MPI_FLOAT;
array_of_types[2] = MPI_INT;
```

- Which requires less typing than it otherwise might!

Describing the Structure

- Having built up these arrays we can finally describe the structure to MPI, which fills in a datatype array

```
MPI_Datatype MyNewType;

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

- The first parameter ('2') indicates the number of entries in each array.
- Finally we 'commit' the datatype, which tells MPI about it

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

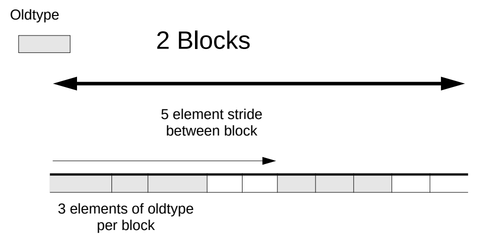
- We can now use `MyNewType` anywhere we would otherwise use an intrinsic type.

Describing a Vector

- Fortunately describing a vector to MPI is slightly simpler!
- Imagine the case where you have a long sequence of data items stored in memory (such as in a matrix) and you wish to extract certain items from it.
- You could copy these out manually into another array, and send this.
- To save you the effort of doing this MPI allows you to define a vector datatype, which includes the extraction information!
- Unfortunately this means you have to describe the datatype yourself.

A Vector

- A vector, in MPI terms appears as:



- The datatype description is generated by

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

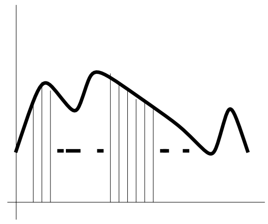
- In the above example blocklength = 3, stride = 5 and count = 2.
- Obviously the vector is useful for extracting elements from matrices.

Other Types

- MPI allows other datatypes to be defined, but for the purposes of this introductory course we will skip the details.
- The vast majority of MPI programs will only derive datatypes which correspond to either structures or vectors, so this isn't too critical!

Exercise 2

- Now, to reinforce how MPI programs send messages, we want to write an application which performs a very simple integration function in parallel.



- To help focus on the message passing part of the problem we provide skeleton programs which can already do most of the calculations, all you need to do is coordinate them!

Cont.

- The integration is performed by a simple Newton-Raphson approximation. The parameter range is split up into strips.
  - The integrate range operation is coded into the slave function. This subroutine has functions which enable it to integrate a number of strips, of fixed width, starting at a particular value. Note that the 'function' is hard-coded in, we never pass it about.
  - The master function reads in the number of processors the user wants, splits the strips into groups for each processor.
- We already provide a skeleton function which splits the range selected by the user into the required number of strips, and calculates which ones to send to which slave.
- You must write the message passing code to actually send the information.

Cont.

- As we have mentioned before the master process sends a derived datatype to the workers.
- We have already derived the type for you, but you might want to examine the code to see what you place in the structure.
- You must write the code to:
  - Send the relevant structure from the master to the workers.
  - Receive this structure on the worker.
  - Send the resulting integration of the sub-range from each worker to the master.
  - Receive each integration on the master.

Conclusions

Course Summary

- This course has covered:
  - The basic concept of a message passing library
  - The background to MPI
  - How MPI 'has learnt' from other library
  - The philosophy of programming in MPI
  - How applications are created using MPI (SPMD)
  - How messages are sent and received by programs operating with MPI
  - Deriving datatypes for MPI.
- MPI is an extremely large standard, we have left out quite a lot of detail, including group operations, profiling, global reduction.

The state of MPI

- MPI is an extremely new standard, and is likely to improve in the future, but at the moment it has 2 principal problems:
  - Public domain implementations tend to be poorly documented, and not very efficient. There are few commercial implementations as yet, so this is hampering MPI.
  - MPI leaves out of the standard some very important details, notably spawning and process management. This limits the applicability of MPI in areas like workstation clusters.
- At the time of writing (May '94) MPI is more suited for use with applications which will have a long life....if you need results in 6 months time you may be better off using PVM.