Parallel Computing
CM30225

Russell Bradford

2023/24

### 1. MPI

In the batch file, mpirun sets up the processors and processes involved

Depending on the MPI implementation, this might be clever and sort out the best transport between them, e.g., in memory for processors on the same node and on the network for processors on different nodes

Or it might simply use network connections, regardless

The programmer uses the same MPI functions to send messages whatever the underlying mechanism

### 2. MPI

#### One-to-one messaging

MPI is about sending messages between processes

A basic use scenario is when one processor wants to send a message (some data) to another



Simple message send

Processor A sends data (integers, floats, strings, etc.) to B

A can use a *send* function, while B uses a *receive* function

### 3. MPI

#### One-to-one messaging

int n[5];
...
if (myrank == 0) {
 MPI\_Send(n, 5, MPI\_INT, 1, 99, MPI\_COMM\_WORLD);
}
else if (myrank == 1) {
 MPI\_Status stat;
 MPI\_Recv(n, 5, MPI\_INT, 0, 99, MPI\_COMM\_WORLD, &stat);
}

We suppose A has rank 0, B rank 1 in WORLD

### 4. MPI

#### One-to-one messaging

MPI\_Send uses

* n A pointer to a memory location containing the data; can be a single variable or (more likely) an array of values
* 5 The number of items to send
* MPI\_INT The type of the items
* 1 The rank of the destination
* 99 A *tag* As there can be many messages flying around you can tag them with specific integers. This allows you match up a particular send with a particular receive
* MPI\_COMM\_WORLD The rank is within this communicator

### 5. MPI

#### One-to-one messaging

MPI\_Recv uses

* n A pointer to a memory location where to store the data: it need not be the same place as A (n in our example) as B is a separate process
* 5 The number of items to read
* MPI\_INT The type of the items
* 0 The rank of the source
* 99 The *tag* on the message you are waiting for: use MPI\_ANY\_TAG if you don’t care
* MPI\_COMM\_WORLD The communicator
* stat A structure contains the status of the transfer, in particular the source and tag; and the error type in case of an error

### 6. MPI

#### Messaging Types

Types include
MPI\_CHAR, MPI\_SHORT, MPI\_INT, MPI\_LONG, MPI\_FLOAT, MPI\_DOUBLE, MPI\_BYTE
among several others

### 7. MPI

#### Messaging Types

MPI\_Send and MPI\_Recv are *blocking*, meaning MPI\_Send waits until the data has been copied out of the buffer n into the messaging subsystem. The array n in A can be safely reused immediately after the MPI\_Send call returns

Note the data itself may not yet have reached or have been read by B

Or even sent yet by A; all we know is that is has been copied out of n

Naturally, MPI\_Recv waits until the data is safely copied into its buffer

### 8. MPI

#### Messaging Types

This provides a weak synchronisation between A and B

All we know is that B has to wait for A: nothing more than that

B gets the data after A produced it

Beyond this synchronisation we can say little about what the relationship between A and B is

For example, A won’t know when B actually gets the data; B doesn’t know when A sent the data

### 9. MPI

#### Asynchronous messaging

In a distributed system you have to be aware of the *asynchronous* nature of communication

As messages take a significant time to be transmitted a send and a receive are certainly non-simultaneous

In comparison, in a shared memory system, once a value is written to a variable, that value is available essentially instantly everywhere (ignoring caching and speed of light issues!)

### 10. MPI

MPI also provides

* MPI\_Ssend Waits until the destination has started to receive the message: a stronger synchronisation, not often needed
* MPI\_Isend Send, but don’t wait and carry on processing. A separate thread or DMA subsystem will asynchronously copy out and send the data. You have to be careful about reusing the buffer too soon (“I” for “immediate”)
* MPI\_Irecv Indicate a buffer where data should be read into, but don’t wait for it; the data will be copied asynchronously into the buffer at some point in the future
* MPI\_Wait Block until a given non-blocking send or recv has completed

And lots more

### 11. MPI

#### Synchronisation

Simple synchronisation can be achieved by MPI\_Barrier(MPI\_Comm comm);

This blocks until all the processes in the communicator have reached the barrier

Note that the processes involved in the barrier are specified by the communicator; compare with pthread barriers that wait for any $n$ threads that happen to arrive

MPI\_Barrier is rarely needed as (a) many of the other MPI functions (MPI\_Send, MPI\_Recv etc.) also synchronise already and (b) SPMD programs generally have less of a need for barriers anyway

If you find yourself using MPI\_Barrier, think again!

### 12. MPI

A quick note on messages:

Messages in MPI are *reliable*, *in order*, but *not fair*

Reliable: messages don’t get lost in the network

In order: if A sends message 1 then message 2 to B, then B will get message 1 before message 2: messages from one source to the same destination do not overtake each other

However, a message from A to B may be overtaken by a later message from C to B: there is no guarantee of order on messages from different sources (e.g., A to B is over the network, but C to B is in shared memory)

### 13. MPI

As usual, “not fair” means “not guaranteed fair”. Mostly things will happen in the expected orders, but you should not rely on it

If you need a specific order, use tags

A blocking receive with a tag will wait until a message with that tag arrives, even if other messages are ready waiting

### 14. MPI

#### Multiple participant messaging

The above send and receive are point-to-point messages, namely one source and one destination

MPI provides many more general kinds of messaging

Point-to-point turns out to be much less useful than you might think

### 15. MPI

Broadcast:
MPI\_Bcast(void\* buffer, int count, MPI\_Datatype datatype, int root, MPI\_Comm comm);

The buffer of data is sent from the process with rank root to *all* processes in the communicator



MPI broadcast

### 16. MPI

Note: all processes, including the receivers, should call MPI\_Bcast with the same value for root

The destination buffer can be different on each processor, but is typically the “same” buffer (in an SPMD sense)

### 17. MPI

int n[2];
if (myrank == 1) {
 n[0] = 23;
 n[1] = 42;
}
...
MPI\_Bcast(n, 2, MPI\_INT, 1, MPI\_COMM\_WORLD);

All processes will now have the same values for their versions of n

### 18. MPI

MPI\_Scatter(void\* sendbuf,int sendcount, MPI\_Datatype sendtype, void\* recvbuf, int recvcount, MPI\_Datatype recvtype, int root, MPI\_Comm comm);

This takes the data sendbuf, an array, in processor with rank root, and sends sendcount items from the array to each other processor (and to itself) to end up in recvbuf



Scattering single values

### 19. MPI

The processor with rank 0 (in the specified communicator) gets the first sendcount items from sendbuf; processor 1 gets the next sendcount items; and so on

Just as in broadcast, every processor executes SCATTER with the same root

Note: recvtype can be different from sendtype, but you had better be sure you understand what you are doing

recvcount can be different from sendcount, but you had better be sure you understand what you are doing

Don’t do that!

### 20. MPI

MPI\_Gather(void\* sendbuf, int sendcount, MPI\_Datatype sendtype, void\* recvbuf, int recvcount, MPI\_Datatype recvtype, int root, MPI\_Comm comm);

Takes sendcount elements of data sendbuf from each processor and puts them in the array recvbuf on processor root



Gathering single values

### 21. MPI

MPI\_Gather is the “opposite” of MPI\_Scatter

The recvbuf on the root processor is filled, in order, with the specified number of items from processors rank 0, 1, etc.

Type and counts can vary across processors

But don’t do that

### 22. MPI

MPI\_Reduce(void\* sendbuf, void\* recvbuf, int count, MPI\_Datatype datatype, MPI\_Op op, int root, MPI\_Comm comm);

Applies a reduction of operation op to each value in sendbuf, putting the result(s) into recvbuf on processor root



MPI reduce

### 23. MPI

Operations include
MPI\_MAX, MPI\_MIN, MP\_SUM, MPI\_PROD, MPI\_LAND (logical AND), MPI\_LOR (logical OR)
amongst others

You can also define your own reduction operators

### 24. MPI

MPI\_Scan(void\* sendbuf, void\* recvbuf, int count, MPI\_Datatype datatype, MPI\_Op op, MPI\_Comm comm);

A *prefix scan* of the source sendbuf. Processor of rank $i$ gets the reduction of values from processors $0…i$ stored in its recvbuf



MPI scan

Prefix scans turn out to be a very useful tool in parallel algorithms

### 25. MPI

As usual with MPI, there are many other combinations of blocking and non-blocking messages possible

Note these functions are **not cheap**: they hide a lot of messaging, which you should be aware of when you are using them

For example, a MPI\_Bcast of a large datastructure can be very slow

### 26. MPI

For timing, MPI\_Wtime() returns a “high precision” elapsed time in seconds on the calling processor

It returns a double, with precision as given by MPI\_Wtick()

This might be, say, 0.000001 (1 microsecond)

### 27. MPI

MPI also provides

* defining new MPI datatypes including arrays and structures;
* means of creating communicators;
* processor groups (communicators contain one or more groups);
* processor topologies (ways of arranging processors into particular geometric shapes that might fit a certain problem or hardware);
* more kinds of scatter/gather/reduce/scan;
* all-to-all broadcasts;
* and so on

### 28. MPI

MPI is used extensively out there in the big world of Real Science

It is very well suited for when there is so much computation needed that the overhead of a bunch of messages is well worth paying

The large (100k core) clusters will be running jobs using MPI

MPI scales very well to large systems

### 29. MPI

And, of course, you can mix shared and distributed memory: running shared memory OpenMP tasks communicating across nodes via MPI

Don’t use OpenMP in the coursework: that should be pure MPI

### 30. MPI

MPI requires you to make sure all your MPI function calls are coordinated across the processes: every processor must call the appropriate same or matching functions at the appropriate times

This the programmer’s problem: it’s a bug if you get it wrong

### 31. MPI

For example, you can still easily deadlock. Suppose A and B wish to exchange messages:

A B
MPI\_Recv(...); MPI\_Recv(...);
... ...
MPI\_Send(...); MPI\_Send(...);

This is slightly more obvious when it happens since MPI is SPMD and has a single program source

Careful use of message tags helps structuring

As is common, MPI provides easy mechanism but no analysis

### 32. MPI

In fact, for this case, MPI provides MPI\_Sendrecv which combines a send with a receive that is guaranteed not to deadlock

A B
MPI\_Sendrecv(...); MPI\_Sendrecv(...);

This function is recommended in cases of swapping data

And it can connect any pair of processes; is not limited to simple swapping between two processes. For example, A sends to B but receives from C; while B sends to C but receives from A; etc.