Lecture 1

In message passing model, parallelism is achieved by having many rocesses cooperate on the same task, each process has its own data and processes communicate via messages only.

A synchronous send is not completed until the message starts to be received.

An asynchronous send completes as soon as the message has gone

Receives are usually synchronous, receiving process must wait until the message arrives before it starts to receive.

Lecture 2

MPI controls its own internal data structures, MPI releases hand les to allow programmers to refer to these. C handles are defined typedefs.

MPI\_Init must be the first procedure called in main and int MPI\_Finalize() must be the last.

int rank;

...

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

printf(“Hello from rank %d\n”, rank);

rank and size are dfined above and below

MPI\_Comm\_size(MPI\_Comm comm, int \*size)

Lecture 3

The compiler used is mpicc

Runnign interactively will not give reliable timings as shared with other users, and there is many more processes that processors, bt is useful for debugging. To run on 4 processes:

mpiexec –n 4 ./mpiprog.exe

To run on Morar you must make a copy of an sge file and rename it to match your executable, except with a .sge format. To run of 4 processors

qsub -pe mpi 4 hello.sge

By default MPI wrappers are not on your path, must type module load PrgEnv-pgi

And add this to your home directory

To compile on archer use cc . Archer usually has substantial queues on the back end. However sometimes the course gets dedicated queues. Archer must have files names ending in .pbs after a name which matches your script. This can be found in templates. To submit work to archer:

qsub –q hello.pbs

MPI is not an OO interface, however, can be called from c++

Lecture 4

There are two MPI data types: basic types and derived types. Derived types can be built from basic.

**MPI Datatype C datatype**

MPI\_CHAR signed char

MPI\_SHORT signed short int

MPI\_INT signed int

MPI\_LONG signed long int

MPI\_UNSIGNED\_CHAR unsigned char

MPI\_UNSIGNED\_SHORT unsigned short int

MPI\_UNSIGNED unsigned int

MPI\_UNSIGNED\_LONG unsigned long int

MPI\_FLOAT float

MPI\_DOUBLE double

MPI\_LONG\_DOUBLE long double

Point to point communication: Communication between two processes. Source sends message to destination process and all communication takes place within the comumunicator. Destination process is identified by its rank in the communicator

Sender calls a send routine, specifying that the data is to be send – this isc called the buffer

Receiver calls a recv routine, where the incoming data should be stored , this is called a receive buffer.

There are a variety of different communication modes:

**Synchronous send** Only completes when the receive has completed

**Buffered send:** always completes (unless an error occurs) irrespective of the receiver

**Standard send**: Either synchronous or buffered

**Ready send:** Always completes ( unless an error occurs) irrespective of whether the reveive has completed

**Receive:** completes when a message has arrived.

**OPERATION MPI CALL Standard send**

MPI\_Send Synchronous send

MPI\_Ssend Buffered send

MPI\_Bsend Ready send

MPI\_Rsend Receive

MPI\_Recv

Sending an array of 10 integers

int x[10];

...

if (rank==1)

MPI\_Ssend(x, 10, MPI\_INT, dest=3, tag=0, MPI\_COMM\_WORLD);

Receiving an array of ten integers

int y[10];

MPI\_Status status;

...

if (rank==3)

MPI\_Recv(y, 10, MPI\_INT, src=1, tag=0, MPI\_COMM\_WORLD, &status);

Processes synchronous if send specifies the synchronous mode. B

**Blocking message passing: both processes wait until the transaction has completed**

The receivers buffer must be large enough for communication to succeed as well as matching tags, matching messages, destination rank, source rank, tags and the same communicator.

Receiver can wildcard: To receive from any source or tag you can write

MPI\_ANY\_SOURCE and MPI\_ANY\_TAG

Actual source and tag are returned in the receivers status parameter.

Messages never over take each other, even for non-synchronous sends

Ibntroducing timers!!!

double MPI\_Wtime(void);

Time to perform a task is measure by consulting the timer before and after. Time is measured in seconds

Lecture 6

MPI\_Ssend: Guaranteed to be synchronous, routine will not return until message has been delivered

MPI\_Bsend: Guaranteed to be asynchronous, routine returns before the message has been delivered. System copies data onto buffer and sends it later on.

MPI\_Send: May be implemented as synchronous or asynchronous send.

Recv: Always synchronous. If process B issued recv before Bsend from process A, then B would wait on the recv until Bsend was issued.

For Bsend, the user provides a large block of memory for buffer space, which is made available to MPI using MPI\_Buffer\_attach If A issues another Bsend before the Recv the system tries to store the message in the free space in the buffer, if there is not enough space then Bsend will fail.

Ssend runs the risk of deadlock, Bsend less likely to dead lock, and your code will run faster but the user must supply the buffer space and the routine will fail if this buffering is exhausted.

MPI\_Send tries to rectify these problems. Buffer space is provided by the system, Send is normally asynchronous, but if the buffer is full, instead of collapsing, send becomes synchronous. MPI\_Send is unlikely to fail, but could cause deadlock if buffer space runs out.

Send(x,B) Send(y,A)

Recv(x,B) Recv(y,A)

This ^^ will dead lock if Send is synchronous (Due to both waiting on receive to receive their sent message) However this is guaranteed to deadlock if you use Ssend.

To avoid deadlock, can match sends and receives explicitly A sends and receives and V receives and sends, EG pingpong.

MPI Allows you to check if any messages have arrived, you can probe for matching messages. Same syntax as receive except no receive buffer specified.

int MPI\_Probe(int source, int tag, MPI\_Comm comm, MPI\_Status \*status)

Status is set as if the receive took place. But BE CAREFUL with any wild cards eg using MPI\_ANY\_SOURCE call when probing. Use a specific source to guarantee matching same message.

Every message can have a non negative tag. Max value can be found using MPI\_TAG\_UB

Attribute. Its perfectly fine to set all tags to 0 usually.

All MPI communications take place within a communicator. A communicators is fundamentally a group of processes, there is a predefined communicator MPI\_COMM\_WORLD which contains all the processors and MPI\_COMM\_SELF which contains only one process.

A message can only be received with the same communicator from which it was sent. It is not possible to wildcard

We can split MPI\_COMM\_WORLD into pieces, each process gets a new rank with each of the sub communicators. Guarantee messages from different pieces do not interact. This can be done using tags, however there are no guarantees.

Can make a copy of MPI\_COMM\_WORLD calling the MPI\_Comm\_dup routine. This contains all the same processes but in a new communicator.

Enables processes to communicate with each other safely within a piece of code. Guaranteed thatmessages cannot be received by other code which is essential for people writing parallel libraries (FFT) to stop library messages from vetting mixed up. (User cannot intercept the library messages if the library keeps the identity of the new communicator a secret, no safe to simply recerve tag values due to wild carding)

Tags can be useful for debugging

Lecture 7

Dead lock can happen when all the systems are waiting on their message to receive and non of them are actually receiving. The mode of communications determines when its constituent operations complete ie synchronous/asynchronous

The form of an operation determines when the procedure implementing that operation will return (ie control returned to the user program)

In my words, the mode determines when the operation is complete, ie does it need to wait on a recv etc. The form determines whether it needs to wait on completion to move forward.

**Blocking operations** do need to wait until that operation is completed before moving on. Only return from the subroutine call when the operation has completed. The routines we have used so far are MPI\_Ssend and MPI\_Recv

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

All non blocking operations should have matching wait operations, some systems cannot free resources until wait has been called. The wait just checks that the sub routine call has completed, and if a non blocking operations is immediately followed by a wait, then it is a blocking operation as blocking operations don’t move on until theyre sure the operation has completed.

Non blocking operations are not the same as sequential subroutine calls as the operation continues after the call has returned.

We can separate nonblocking communication into three phases:

1. Initiate non blocking communication
2. Do some work (Perhaps involving other communications)
3. Wait for non blocking communication to complete.

Datatypes same as for blocking, and communicatior the same as for blocking. Request MPI\_Request or integer. A request handle is allocated when a communication is initiated.

A non blocking Synchronous send and then the following wait is :

int MPI\_Issend(void\* buf, int count,

MPI\_Datatype datatype, int dest,

int tag, MPI\_Comm comm, MPI\_Request \*request)

int MPI\_Wait(MPI\_Request \*request, MPI\_Status \*status)

And a non blocking synchronous send and recv are:

int MPI\_Irecv(void\* buf, int count, MPI\_Datatype datatype,

int src, int tag, MPI\_Comm comm, MPI\_Request \*request)

int MPI\_Wait(MPI\_Request \*request, MPI\_Status \*status)

Send and recv can be blocking or nonblocking. A blocking send can be used with a non blocking recv, and vice versa. Non blocking sends can use any mode : Synchronous, buffered, standard or ready. Synchronous mode affects completion, not initiation.

You can test or wait for completion, testing is done as follows

int MPI\_Test(MPI\_Request \*request, int \*flag, MPI\_Status \*status)

Can test or wait for one message, all messages or as many messages as possible.

You can specify send and recv arguments in one call. MPI implementation avoids deadlock, useful in simple pairwise communication patterns, but not as generally applicable as non blocking.

int MPI\_Sendrecv(void \*sendbuf, int sendcount, MPI\_Datatype sendtype,

int dest, int sendtag, void \*recvbuf, int recvcount,

MPI\_Datatype recvtype, int source, int recvtag,

MPI\_Comm comm, MPI\_Status \*status);