THE UNIVERSITY *of* York

High Performance Computing - MPP Programming with MPI

Prof Matt Probert

http://www-users.york.ac.uk/~mijp1

Overview

- Basic Ideas
- Point-to-Point Communication
  - blocking version
- Simple Collective Communication

Basic Idea

- MPI is a portable library
  - Flexible, powerful, easy to use, very general application
  - Implementer can customise internals to machine architecture
- SPMD paradigm
  - Single Program, Multiple Data
    - NB Not single instruction – code may branch so that different processors do different jobs – task farming – or work on different copies of the data
    - Distributed memory architecture

Key Concepts I

- Messages
  - Transfers data between processors:
    - Which processor is sending the message?
    - Where is the data on the sending processor?
    - What is the “type” of the data?
    - How much data is to be sent?
    - Which processor(s) are to receive the data?
    - Where should the data be put on the receiver?
- Communicators
  - A way of dividing up the available processors into separate groups that can then co-operate on a task
  - All message passing is within a communicator so messages in different communicators cannot clash
  - Sounds complex but it greatly simplifies coding!

Key Concepts II

- Handles
  - All this sounds very complicated but it is not as the details are hidden within the library and so are up to the implementer not the MPI programmer!
  - All structures are referenced by *handles* – simple integers – which reference an entry to a table inside the MPI library
- C and FORTRAN Support
  - All MPI routines and constants begin with MPI_ to avoid clashes with other libraries. FORTRAN is case insensitive but in C (case sensitive) names are mixed case and all constants are upper case.
  - The C-versions return an `int` and FORTRAN have an `IERROR` parameter to return error codes
    - `MPI_SUCCESS` (zero) indicates success. Numeric codes are non-portable so use `MPI_ERROR_STRING` routine to translate into text.

Key Concepts III

- Types of MPI routine:
  - System query
    - Who am I? Finding out about process id, etc.
  - Point-to-Point
    - Send/receive pairs in different variations
  - Collective
    - One-to-all, all-to-all, all-to-one and barriers
  - Miscellaneous
    - MPI derived data-types, communicator management, error handling, start-up and shut-down.

Hello World

- The classic first program to write:

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

int main(int argc, char **argv)
{
   /* Initialise MPI */
   MPI_Init (&argc, &argv);

   printf("Hello world \n");

  /* Terminate MPI */
   MPI_Finalize ();

   exit (0);
}
```

```fortran
program hello
  use mpi
  implicit none

  integer :: ierror

  call MPI_Init(ierror)

  print *,"Hello World"

  call MPI_Finalize(ierror)

end program hello
```

NB Ought to check status of error code after each call

NB C/C++ is case-sensitive – use CamelCase – but Fortran is not!

Who am I?

One of the first tasks, once MPI has been initialised, is to find out how many processes there are …

`call MPI_comm_size(MPI_comm_world,size,ierror)`

where `MPI_comm_world` is a handle to the default communicator – the set of all available processes – defined in `‘use mpi’` or `‘#include mpi.h’`

and `size` is int num of processes in communicator.

`call MPI_comm_rank(MPI_comm_world,rank,ierror)`

gives `rank` (process id number) in range `0 ≤ rank ≤ size-1`

Who am I? (F90 version)

```
program hello
  use mpi
  implicit none
  integer :: ierror, myrank, size

  call MPI_Init(ierror)

  call MPI_Comm_rank(MPI_comm_world,myrank,ierror)

  call MPI_Comm_size(MPI_ comm_world,size,ierror)

  print *,”Hello World from processor”,myrank+1,”of”,
size

  call MPI_Finalize(ierror)

end program hello
```

Who am I? (C version)

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

void main (int argc, char *argv[]) {
  int myrank, size;

  MPI_Init(&argc, &argv);         /* Initialize MPI */
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);/* get rank  */
  MPI_Comm_size(MPI_COMM_WORLD, &size); /* get size */

  printf("Processor %d of %d: Hello World!\n", myrank,
size);

  MPI_Finalize();                  /* Terminate MPI */
}
```

What can we send?

We can send single values (or arrays) of the following types.

MPI Datatype	C Datatype
MPI_CHAR	Signed char
MPI_SHORT	Signed short int
MPI_INT	Signed int
MPI_LONG	Signed long int
MPI_UNSIGNED_CHAR	Unsigned char
MPI_UNSIGNED_SHORT	Unsigned short int
MPI_UNSIGNED	Unsigned int
MPI_UNSIGNED_LONG	Unsigned long int
MPI_FLOAT	Float
MPI_DOUBLE	Double
MPI_LONG_DOUBLE	Long double
MPI_BYTE	
MPI_PACKED	

MPI Datatype	FORTRAN Datatype
MPI_INTEGER	Integer
MPI_REAL	Real
MPI_DOUBLE_PRECISION	Double precision
MPI_COMPLEX	Complex
MPI_LOGICAL	Logical
MPI_CHARACTER	Character(1)
MPI_BYTE	
MPI_PACKED	

NB Fortran is F77-style, i.e. no F90-style `kind` parameters.

`MPI_BYTE` is for 8-bit data.

`MPI_PACKED` is for later …

Communication Types

- Synchronous
  - Operation does not complete until message has been received – c.f. sending a fax message
- vs. Asynchronous
  - Operation completes as soon as message is on its way – c.f. posting a letter
- Blocking
  - Operation only returns from subroutine when operation has been completed – c.f. fax machine without memory – stays busy until message is sent and cannot send another one in the mean-time.
- vs. Non-Blocking
  - Operation returns immediately and allows program to continue with other operations – c.f. turning on a fax machine to receive a message

Point to Point Communication

- Consider pseudo code for point to point communication, e.g. CPU 0 sends to CPU 1:

```
if myrank = 0 then
        call an MPI send routine
else if myrank = 1 then
        call an MPI receive routine
end if
```

- Other CPUs don’t do anything – may get ‘ahead’ of the first two.

- Many options for how and when the send/receive occurs.

Sending Options

- The task which `send`s calls one of the following routines:

Mode	Blocking	Non-Blocking
Standard	`MPI_Send`	`MPI_Isend`
Buffered	`MPI_Bsend`	`MPI_Ibsend`
Synchronous	`MPI_Ssend`	`MPI_Issend`

- `SEND` uses either `BSEND` or `SSEND` – depends on MPI implementation – **simplest to use!**
- `BSEND` buffers messages so can transmit several in one go at some later time – completes when the message has been buffered.
- `SSEND` sends the message and will not send another until it has been received – completes when the message has been received.

Blocking or Non-Blocking?

- *Completion condition* is different for each type of send.
- Blocking versions of routines return when the operation has satisfied the completion condition.
- Non-blocking versions return when the operation has begun.
  - Why? So can hide comms latency with non-blocking versions by performing computation whilst comms in progress => more efficient

Receiving a Message

- Regardless of the sending mode, the receiving CPU calls the same routines:
  - `MPI_Recv` (blocking)
  - `MPI_Irecv` (non-blocking)
- In the non-blocking case, we cannot assume the message has been received (and hence use the data received) until we check for completion:
  - `MPI_Wait` stops execution until message has been received.

  There is also `MPI_Test` which checks if the task (`send` or `recv`) has completed but does not wait if it has not.

Standard Blocking Send

```
call MPI_Send(data, count, datatype, dest, &
              tag, comm, ierr)
```

F90

```
ierr = MPI_Send(&data, count, datatype, dest,
                tag, comm);
```

C

- `data` is the address of the data to be sent (e.g. variable name if scalar, or first element of 1D or 2D array),
- `count` is the number of elements of MPI `datatype` within `data` (e.g. `n*n` for a NxN array)
- `dest` is the destination, i.e. rank of the receiving process which must be within the same `comm`unicator,
- `tag` is a marker for the programmer to distinguish different types of message
- `ierr` is error code (0=success).

Standard Blocking Receive

F90

```
call MPI_Recv(data, count, datatype, source, tag, &
                 comm, status, ierr)
```

C

```
ierr = MPI_Recv(&data, count, datatype, source, tag,
                   comm, &status);
```

`data` is the address of the data to be placed once received (e.g. variable name if scalar, or first element of an existing array which must be large enough!)

`count` is the number of elements of MPI `datatype`

`source` is the rank of the sending process which must be within the same `comm`unicator.

`tag` must match that in specified in the send unless `MPI_ANY_TAG` wildcard used instead

`status` is a special handle to be interrogated later…

`ierr` is the return value (0=success).

Comments

- A sender can "push" a message but a receiver cannot "pull" – can only fetch a message already "out there".
  - MPI is the middle-man: sender posts a message, receiver posts a matching receive, and MPI joins up
  - Tags enable receiver to choose which message to receive *before* the receive begins.
  - `status` is an integer array (of `MPI_STATUS_SIZE`) which **must** be declared by user code and holds information about message,
    - `status(MPI_SOURCE)` gives rank of sender
    - `status(MPI_TAG)` gives tag of message
    - `MPI_GET_COUNT(status, datatype, count)` gives number of elements of data actually received in count.
    - A special datatype MPI_Status is provided in C/C++/F2008.

Simple Blocking Send/Recv

e.g. process 0 sends an array ‘a’ with 100 elements to process 1

```
! Process 0 sends, process 1 receives:
if (myrank == 0) then
  call MPI_Send(a(1),100,MPI_DOUBLE_PRECISION,1,17,MPI_COMM_WORLD,ierr)
else if (myrank == 1) then
  call MPI_Recv(a(1),100,MPI_DOUBLE_PRECISION,0,17,MPI_COMM_WORLD, &
               & status,ierr)
endif
```

```
/* Process 0 sends, process 1 receives: */
if ( myrank == 0 ) /* Send a message */
  ierr = MPI_Send( &a[0], 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
else if ( myrank == 1 ) /* Receive a message */
  ierr = MPI_Recv( &a[0], 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD,
&status );
```

(should compare `ierr` to MPI_SUCCESS after each call)

Swapping information?

```
/* DO NOT DO THIS */
if (myrank == 0) {
/* Receive, then send a message */
   MPI_Recv( &b[0], 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD,
&status );
   MPI_Send( &a[0], 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
}
else if (myrank == 1) {
/* Receive, then send a message */
   MPI_Recv( &b[0], 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD,
&status );
   MPI_Send( &a[0], 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
}
```

- Rank 1 cannot send until it receives from rank 0
- Rank 0 cannot send until it receives from rank 1
- Serious chicken and egg situation – DEADLOCK!

Deadlock

- Deadlock is the bane of parallel programs:
  - A program that appears to runs fine on one implementation of MPI fails on a different one or may randomly "hang".
  - Usually caused by a communication mismatch, e.g. one process is waiting for a message that will never come, etc.
  - Or chicken + egg situation as previous slide.
  - Must guarantee that all messages sent will eventually be received else will overload the comms network.
  - Blocking sends causes synchronisation and hence potential for deadlock – non-blocking sends can eliminate this – see next MPI lecture.

Point-to-Point Semantics

- Message order is preserved
  - If A send two messages to B, and B posts two matching receives, then they will be received in the order they were sent.
- Progress
  - It is not possible for a matching send and receive pair to remain permanently outstanding
    - Either send or receive will eventually complete:
    - A third process posts a matching receive in which case the send completes but not the receive, or
    - A third process sends out a matching message which is received instead, hence the second process receive completes but not the first process send.
- Datatypes must match in send and receive and with corresponding language type of data, except for `MPI_PACKED`.

Basic Collective Communication

- `MPI_Bcast(data, count, datatype, root, comm, ierror)`
  - Broadcasts `count` items of `data` from `root` process to all process in specified `comm`unicator.
  - All non-root nodes receive data – no need for a matching receive command.
- NB Collective communications are transparent to point-to-point and v.v. – no problem with clashing.
- NB No tags and no part-full buffers allowed.
- NB No non-blocking collectives in MPI v1 – command completion implies data may be reused.

Miscellaneous MPI Commands

- `MPI_Wtime()` – a simple function that returns a double precision wall-time in seconds
  - Hence need a pair of calls to time a chunk of code
- `MPI_Wtick()` gives timer resolution as double precision (e.g 10^{-3} means millisecond resolution).
- `MPI_Barrier(comm, ierror)`
  - A synchronisation command – all processes in `comm`unicator will wait until all reach the barrier – hence all must call it!

MPI summary

- MPI is a large and flexible library (125 functions in MPI v1 & more in v2 and v3)
- But actually you only need to know 6 functions to write many programs:
- `MPI_Init, MPI_Finalize`
- `MPI_Comm_size, MPI_Comm_rank`
- `MPI_Send, MPI_Recv`

Next MPI Lecture

- Non-blocking point to point communication.
- More advanced collective communication.
- Advanced communicators and topologies.

Further Reading

- Chapter 9 of "Introduction to High Performance Computing for Scientists and Engineers", Georg Hager and Gerhard Wellein, CRC Press (2011).
- "Using MPI, 2nd edition", William Gropp *et al*, MIT press (1999).
- EPCC course notes at http://www.epcc.ed.ac.uk/education-training/
- MPI forum https://www.mpi-forum.org
- MPI homepage http://www.mcs.anl.gov/research/projects/mpi including MPI standards, examples and more.