MPI-2: Message Passing Interface

Nick Dingle and Will Knottenbelt

{njd200,wjk}@doc.ic.ac.uk

Slides available from: `http://www.doc.ic.ac.uk/~njd200`

Recommended Reading

- W. Gropp, E. Lusk and A. Skjellum: “Using MPI: Portable Parallel Programming with the Message-Passing Interface”, 2^{nd} Edn., MIT Press, 1999.
- W. Gropp, E. Lusk and R. Thakur: “Using MPI-2: Advanced Features of the Message-Passing Interface”, MIT Press, 1999.
- G. Karypis, V. Kumar et al. “Introduction to Parallel Computing”, 2^{nd} Edn., Benjamin/Cummings, 2003 (Chapter 6).
- MPI homepage (incl. MPICH user guide): `http://www-unix.mcs.anl.gov/mpi/`
- MPI forum (for official standards): `http://www.mpi-forum.org/`

Outline

- Introduction to MPI-2
- MPI-2 for PC clusters (MPICH-2)
- Basic features
- Non-blocking sends and receives
- Collective operations
- Advanced features of MPI-2

Introduction to MPI-2

- MPI-2 (Message-Passing Interface-2) is a standard library of functions for sending and receiving messages on parallel/distributed computers or workstation clusters.
- C/C++ and Fortran interfaces available.
- MPI is independent of any particular underlying parallel machine architecture.
- Processes communicate with each other by using the MPI library functions to send and receive messages.
- Successor to MPI, incorporating all the functionality of the previous version but adding additional features.
- Over 120 functions in standard; only 6 needed for basic communication.

MPI-2 for PC clusters (MPICH-2) Setup

- Create a file called `.mpd.conf` in your home directory (at `/homes/login`)
- Set the permissions so that only you can read and write this file:
  `% chmod 600 .mpd.conf`
- Enter a secret word into `.mpd.conf`:
  `password=mysecretword`
- Note that you only need to do these two steps once, not every time you wish to compile/run an MPI job.

MPI-2 for PC clusters (MPICH-2) I

- MPICH-2 is installed on the lab machines. The machines `vector01` through `vector10` should always be available for running MPI jobs, with the remainder available outside of lab hours.
- Set up a file called `mpd.hosts`, e.g.
  `vector01.doc.ic.ac.uk`
  `vector02.doc.ic.ac.uk`
- Make sure you can `ssh` to the machines: e.g.
  `ssh vector01.doc.ic.ac.uk uptime`
  (`http://www.doc.ic.ac.uk/csg/linux/ssh.html` has help if this fails).

MPI-2 for PC clusters (MPICH-2) II

- Compile your C program:
  `% mpicc sample.c -o sample`
- Or for C++ source:
  `% mpic++ sample.cxx -DMPICH_IGNORE_CXX_SEEK`
- Boot `mpd` on the machines specified in `mpd.hosts`:
  `% mpdboot -n 4`
- Run your program:
  `% mpiexec -n 4 sample`
- Note that the number of machines you run on does not have to be the same as the number of `mpd` daemons.
- When execution is done, shutdown all `mpd` daemons:
  `% mpdallexit`

Basic features: First and last MPI calls

- **Initialise MPI**:
  `int MPI_Init(int *argc, char ***argv);`

  e.g.:

```
int main(int argc, char *argv[]) {
  if (MPI_Init(&argc,&argv)!=MPI_SUCCESS) {
    ... error ...
  } ...etc...
}
```

- **Shutdown MPI**:
  `int MPI_Finalize(void);`

  e.g. `MPI_Finalize();`

Basic features: The environment

- **Rank identification**:
  `int MPI_Comm_rank(MPI_Comm comm, int *rank);`

  e.g.:

  ```
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  ```

- **Find number of processes**:
  `int MPI_Comm_size(MPI_Comm comm, int *size);`

  e.g.:

  ```
  int size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  ```


A very basic C++ example

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

int main(int argc, char *argv[]){

  int rank, size;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  cout << "[" << rank << "] of " << size << " processors
     reporting!" << endl;

  MPI_Finalize();

  return 0;
}
```


Basic features I

- **Sending a message (blocking)**:

  ```
  int MPI_Send(void* buf, int count,
    MPI_Datatype datatype, int dest,
    int tag, MPI_Comm comm);
  ```

  e.g.:

  ```
  #define TAG_PI 100

  double pi = 3.1415926535;

  MPI_Send(&pi, 1, MPI_DOUBLE, 0, TAG_PI,
    MPI_COMM_WORLD);
  ```


Basic features II

- **Receiving a message (blocking)**

  ```
  int MPI_Recv(void* buf, int count,
    MPI_Datatype datatype, int source,
    int tag, MPI_Comm comm,
    MPI_Status *status);
  ```

  e.g.:

  ```
  double num;
  MPI_Status status;

  MPI_Recv(&num, 1, MPI_DOUBLE,
    MPI_ANY_SOURCE, MPI_ANY_TAG,
    MPI_COMM_WORLD, &status);
  ```

Basic features III

- Receive status information includes:

```
status.count       = message length
status.MPI_SOURCE  = message sender
status.MPI_TAG     = message tag
```

- Note the special tags:

```
MPI_ANY_SOURCE     MPI_ANY_TAG
```


Basic features: Data types

- MPI datatypes include:

```
MPI_CHAR             MPI_BYTE
MPI_SHORT            MPI_INT
MPI_LONG             MPI_FLOAT
MPI_DOUBLE           MPI_PACKED
MPI_UNSIGNED         MPI_UNSIGNED_CHAR
MPI_UNSIGNED_LONG    MPI_UNSIGNED_SHORT
```

- It is possible to create other user-defined datatypes.

A simple C message-passing example

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

int main(int argc, char *argv[])
{
  char msg[20], smsg[20];
  int rank, size, src, dest, tag;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (size!=2) {
    MPI_Abort(MPI_COMM_WORLD, 1);
    return 1;
  }
```


A simple C message-passing example

```c
  src  = 1;
  dest = 0;
  tag  = 999;

  if (rank==src) {
    strcpy(msg, "Hello World");
    MPI_Send(msg, 12, MPI_BYTE, dest, tag, MPI_COMM_WORLD);
  } else {
    MPI_Recv(smsg, 12, MPI_BYTE, src, tag, MPI_COMM_WORLD, &status);
    if (strcmp(smsg, "Hello World"))
      fprintf(stderr, "Message is wrong !\n");
    else
      fprintf(stdout, "Message(%s) %d->%d OK !\n", smsg, src, dest);
  }

  MPI_Finalize();
  return 0;
}
```

Non-blocking sends/receives I

- **Non-blocking send/receive**:

```
int MPI_Isend(void* buf, int count,
  MPI_Datatype datatype, int dest,
  int tag, MPI_Comm comm,
  MPI_Request *request);

int MPI_Irecv(void* buf, int count,
  MPI_Datatype datatype, int source,
  int tag, MPI_Comm comm,
  MPI_Request *request);
```


Non-blocking sends/receives II

- **Wait for send/receive completion**:

```
int MPI_Wait(MPI_Request *request,
  MPI_Status *status);

int MPI_Waitall(int count,
  MPI_Request *array_of_requests,
  MPI_Status *array_of_statuses);
```

- **Non-blocking probe for a message**:

```
int MPI_Iprobe(int source, int tag,
  MPI_Comm comm, int *flag,
  MPI_Status *status);
```

- `flag` is set if message waiting
- `status` has details of message

Collective operations

- Often need to communicate between groups of processes rather than just one-to-one, and MPI defines a large number of collective operations to enable this.
- These groups communicate using specific communicators rather than the message tags used in one-to-one communication.
- Three classes of collective operations:
  - Data movement
  - Collective computation
  - Explicit synchronisation
- Note that all collective operations are blocking operations within the particpating communication group.

Creating your own communicators

- You create your own communicators by splitting up pre-existing communicators:

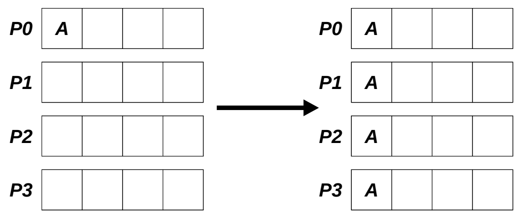
```
int new_group_size = 3;
int new_group_members[] = {1,3,5};
MPI_Group all, some;
MPI_Comm subset;

MPI_Comm_group(MPI_COMM_WORLD, &all);
MPI_Group_incl(all, new_group_size,
  new_group_members, &some);
MPI_Comm_create(MPI_COMM_WORLD, some,
  &subset);
```

- The complementary function, `MPI_Group_excl`, also exists.

Data movement operations I

- **Broadcasting**:




```
int MPI_Bcast(void* buffer, int count,
  MPI_Datatype datatype, int root,
  MPI_Comm comm );
```


Data movement operations II

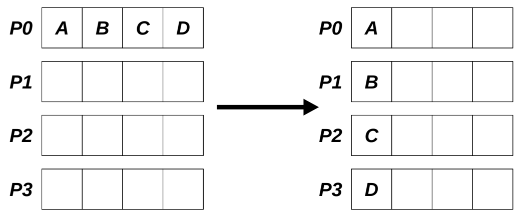
- **Multicasting**:
  Most elegant way is to create a communicator for a subset of the MPI processes, and broadcast to that subset:

```
int new_group_size = 3;
int new_group_members[] = {1,3,5};
MPI_Group all, some;
MPI_Comm subset;

MPI_Comm_group(MPI_COMM_WORLD, &all);
MPI_Group_incl(all, new_group_size,
  new_group_members, &some);
MPI_Comm_create(MPI_COMM_WORLD, some,
  &subset);
MPI_Bcast(buffer, ... , subset);
```


Data movement operations III

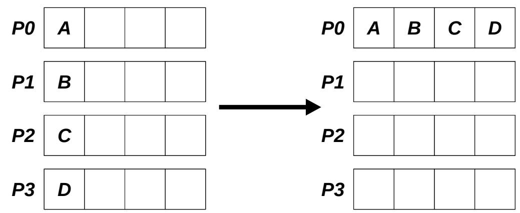
- **Scatter operation**:




```
int MPI_Scatter(void* sendbuf, int sendcount,
  MPI_Datatype sendtype, void* recvbuf,
  int recvcount, MPI_Datatype recvtype,
  int root, MPI_Comm comm);
```


Data movement operations IV

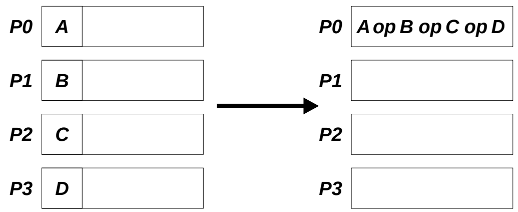
- **Gather operation**:




```
int MPI_Gather(void* sendbuf, int sendcount,
  MPI_Datatype sendtype, void* recvbuf,
  int recvcount, MPI_Datatype recvtype,
  int root, MPI_Comm comm);
```

Collective computation operations

- **Reduce operation**:




```
int MPI_Reduce(void* sendbuf, void* recvbuf,
  int count, MPI_Datatype datatype,
  MPI_Op op, int root, MPI_Comm comm);
```

- Useful ops include `MPI_SUM`, `MPI_PROD`, `MPI_MIN` and `MPI_MAX`.
- Can also define your own operations.

Collective operations

- There are also a large number of MPI collective operations beyond those shown here.
- Those starting with `All` deliver results to all participating processes (e.g. `MPI_Allgather`).
- Those ending with `v` allow different sizes of buffer to be sent and received (e.g. `MPI_Scatterv`).

Explicit synchronisation

- **Barrier synchronization**:
  `int MPI_Barrier(MPI_Comm comm);`
- **Timing your program**:
  `double MPI_Wtime();`

Non-contiguous data

```
int MPI_Pack_size(int incount,
  MPI_Datatype datatype, MPI_Comm comm,
  int *size);

int MPI_Pack(void* inbuf, int incount,
  MPI_Datatype datatype, void *outbuf,
  int outsize, int *position,
  MPI_Comm comm);

int MPI_Unpack(void* inbuf, int insize,
  int *position, void *outbuf, int outcount,
  MPI_Datatype datatype, MPI_Comm comm);
```

Advanced features of MPI-2

- MPI-2 introduces 3 new advanced features:
  - Parallel I/O
  - Remote memory operations
  - Dynamic process management

Parallel I/O

- MPI-1 relied on OS I/O functions, but MPI-2 provides `MPI_File` functions for dedicated parallel I/O:

```
int MPI_File_open(MPI_Comm comm, char *name,
  int mode, MPI_Info info, MPI_File *fh);

int MPI_File_seek(MPI_File fh,
  MPI_Offset offset, int whence);

int MPI_File_read / MPI_File_write(
  MPI_File fh, void *buf, int count,
  MPI_Datatype type, MPI_Status *status);

int MPI_File_close(MPI_File *fh);
```

- Also supports parallel I/O for non-contiguous data, non-blocking parallel I/O and shared file pointers.

Remote memory operations

- Based on *windows* into each process's address space:

```
int MPI_Put / int MPI_Get(void *srcaddr,
  int srccount, MPI_Datatype srctype,
  int targrank, MPI_Aint targdisp,
  int targcount, MPI_Datatype targtype,
  MPI_Win win);

int MPI_Accumulate(void *srcaddr,
  int srccount, MPI_Datatype srctype,
  int targrank, MPI_Aint targdisp,
  int targcount, MPI_Datatype targype,
  MPI_Op op, MPI_Win win);
```

- These operations are non-blocking.
- Note that functions like `MPI_Win_lock()` aren't shared memory locks!

Dynamic process management

- In the MPI-1 standard, the number of processors a given MPI job executes on is fixed.
- In MPI-2 supports dynamic process managment to allow:
  - New MPI processes to be spawned while an MPI program is running.
  - New MPI processes to connect to other MPI processes which are already running.
- Interesting to compare PVM with MPI-1 and MPI-2!