MPI Basics

- Goal: Overview of MPI
- How to compile and run MPI programs
- Main MPI commands - examples
- See many other online resources available

Where to find help

- Many tutorials and other documents including the MPI standard available online
- The openMPI latest documentation
  https://www.open-mpi.org/doc/current/
- The original MPI site:
- The LLNL link:
  https://computing.llnl.gov/tutorials/mpi/

Getting started: hello world from each process

```c
#include <stdio.h>
#include <mpi.h>
int main ( int argc , char * argv []) {
  int nPEs , rank ;
  MPI_Init (& argc , & argv );
  MPI_Comm_size ( MPI_COMM_WORLD , & nPEs );
  MPI_Comm_rank ( MPI_COMM_WORLD , & rank );
  printf ( "Hello from process %d out of %d \n" , rank , nPEs );
  MPI_Finalize ();
}
```

Comments:

- `#include <mpi.h>` is mandatory: contains MPI definitions, types, etc,
- `MPI_Init(.,.)` and `MPI_Finalize()` start and end MPI respectively
- `MPI_Comm_size` gives the number of processes
- `MPI_Comm_rank` gives the number of this process
- Note: all non-MPI functions/commands are local: e.g., `printf` will run on each PE.
Communicators

- A communicator defines a group of processes which communicate with each other.

- [MPI_COMM_WORLD] is the (default) communicator: an internal structure which includes all processes that are allowed to communicate with each other within this run.

- Ranks, sizes, are all relative to a communicator.

- Can define other communicators than MPI_COMM_WORLD by command MPI_Comm.

- Can work with communicators using various commands: MPI_Comm_group, MPI_Group_excl, MPI_Comm_create, ....

The “phixx cluster”

- A small cluster of 16 Xeon processors connected via a network

- Machines in the cluster: phi01, phi02, ..., phi08

- Login to any of these.

- System is dedicated to teaching parallel programming in CSE

Compiling, linking, and running a program

- On the phixx cluster - openmpi version 1.10.2 is loaded by default - but we will use version 1.3

- Instructions will be provided on how to load required modules, compiling, etc

[One note: It is important that *all nodes* run the same version!]

- For compiling: MPI provides mpicc for C and mpif77 (fortran 77), mpif90 (fortran 90 when installed) for fortran.

- For the “hello world” example:

  ```
  mpicc -o hello.ex hello.c
  ```

- For large programs use makefiles

Running a program

- in MPICH, and openMPI a program is run with mpirun:

  ```
  mpirun -np 2 hello.ex
  ```

A few options

- help: print mpi options

  ```
  mpirun -h[elp]
  ```

- use filename as host-file:

  ```
  mpirun -np 4 -machinefile < filename > executable.ex
  ```

- Note you will need a machinefile on the phixx cluster. [a default one will be made available].
Remember: hardest aspect of parallel programming is:

**Debugging**

- explore the use of MCA parameters (Modular Component Architecture) for debugging (and more) in
  [https://www.open-mpi.org/faq/?category=debugging](https://www.open-mpi.org/faq/?category=debugging)

- More information on the class web-site and/or canvas

---

**Sends and Receives**

- Simplest types of communication yet there are several options. For example:
  - Synchronous send
  - Blocking send / blocking receive
  - Non-blocking send / non-blocking receive
  - Buffered send

---

**Blocking send** will complete (return) only when the buffered data has been sent or saved and it is safe to free or reuse send buffer

**Synchronous blocking send:** Handshaking takes place between receiving and sending processes before actual send.

**Asynchronous blocking send:** data buffered until eventual delivery to receiving process

**Blocking Receives** complete (return) after the data has arrived to receiving process

**Non-blocking sends and receives** are similar - they return almost immediately. No waiting for communication to complete

---

**Non-blocking communications are useful when overlapping computation with communication**

- Non-blocking operations “request” the MPI library to perform the operation when possible. Cannot predict when this takes place

- The application buffer (variable space, i.e., array to be sent) should not be changed until it is known that the requested non-blocking operation was performed

- Can use “wait” routines and MPI_Iprobe for this (see later)
**Sends and Receives: Blocking send**

**MPI_Send**(start, count, datatype, dest, tag, comm)

- **start** is a pointer to first entry of data to be sent
- **count** is the length of array
- **datatype** one of MPI's datatypes: MPI_INT, MPI_FLOAT, MPI_DOUBLE, etc,...
- **dest** is the destination process
- **comm** is the communicator
- **tag** is a tag assigned to message so it is recognized by a matching receive.

```c
MPI_Status status;
MPI_Recv(..., &status);
... status.MPI_TAG; <!-- get tag
... status.MPI_SOURCE; <!-- get source
MPI_Get_count(&status, datatype, &count); <!-- get count
```

What is in the MPI.status struct?

- **MPI_SOURCE** - id of processor sending the message
- **MPI_TAG** - the message tag
- **MPI_ERROR** - error status
- **MPI_LENGTH** (Not accessible)
- **MPI_COMM** - communicator,
- Other members reserved for internal implementation.

There should be an associated receive. General form:

**MPI_Recv**(start, count, datatype, source, tag, comm, status)

- Start, count, datatype, comm, have the same meaning as in send
- **source** is the rank of sending process
- **tag** is the same as the tag used in the send
- Can also use: **MPI_ANY_TAG** and / or **MPI_ANY_SOURCE**
- **status** is a struct of type MPI_Status
- The source, tag, and count of the message actually received can be retrieved from status.
Can check that a message has arrived with MPI_Iprobe or MPI_Probe

```c
int MPI_Iprobe(int src, int tag, MPI_Comm comm, int* flag, MPI_Status stat)
```

MPI_Probe is a blocking version of MPI_Iprobe

Can loop-wait until message arrives

Can retrieve the length of a message, with MPI_Get_count:

```c
int MPI_Get_count(MPI_Status *stat, MPI_Datatype dtyp, int *cnt)
```

Sets cnt to number of items in the message.

Example of usage: passing a variable-length string

```c
int source = 0, tag = 100, len;
char *str;
MPI_Status status;
MPI_Probe(source, tag, MPI_COMM_WORLD, &status);
MPI_Get_count(&status, MPI_CHAR, &len);
if(len != MPI_UNDEFINED) str = malloc(len);
MPI_Recv(str, len, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status)
```

Simple collective operations

**MPI_Bcast**

```c
MPI_Bcast(start, count, datatype, root, comm)
```

Sends data from one process to all others.

- start, count, datatype, comm : as before
- root: origin (process #) of broadcast

**MPI_Reduce**

```c
MPI_Reduce(start, result, count, datatype, MPI_OPE, root, comm)
```

**MPI_OPE**: OPE is one of MAX, MIN, SUM, PROD, LAND, BAND, LOR, BOR, LXOR, BXOR, MAXLOC, MINLOC

LAND = logical AND, BAND = bitwise AND. (Similarly for OR, XOR). MPI_MAXLOC, MPI_MINLOC: Find Max(min) and location of max (min)

Scatter and Gather

**MPI_Scatter**

```c
MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtyp, void *recvbuf, int recvcnt, MPI_Datatype recvtyp, int root, MPI_Comm comm)
```

Scatters sendcnt items in succession from sendbuf to processors 0, ..., p – 1.

- Arrays sent to each PE all have same length (sendcnt)

**MPI_Gather**

```c
MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtyp, void *recvbuf, int recvcnt, MPI_Datatype recvtyp, int root, MPI_Comm comm)
```
## MPI Collective communication routines

<table>
<thead>
<tr>
<th>Allgather</th>
<th>Allgatherv</th>
<th>Allreduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alltoall</td>
<td>Alltoallv</td>
<td>Bcast</td>
</tr>
<tr>
<td>Gather</td>
<td>Gatherv</td>
<td>Reduce</td>
</tr>
<tr>
<td>ReduceScatter</td>
<td>Scan</td>
<td>Scatter</td>
</tr>
</tbody>
</table>

- The 'v' versions: Scatterv, Gatherv, Allgatherv, Alltoallv, allow variable counts and shifts on the original array for each destination

### Scatterv:

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
<th>Process 2</th>
<th>Process 3</th>
</tr>
</thead>
</table>

### Appendix: set-up for the phi cluster

- The nodes are called phi01, phi02, ..., phi08
- You can login to any of them via `ssh phi0x.cselabs.umn.edu`
- You need to use mpi version 1.3 [others won't work]
  ```
  module load hpc/openmpi
  ```
- Add this to your init shell file [e.g., in `.cshrc` or `.bashrc`]
- One key requirement is that you should be able to login to any node without requiring a password. Find out how to use ssh-keygen and enable ssh without passwords:
  ```
  ssh-keygen -t rsa -P ""
  cat ~/.ssh/id_rsa.pub > ~/.ssh/authorized_keys
  ```
- You will need to add all the machines into `~/.ssh/known_hosts` file. For this you can ssh once to each machine. This will prevent prompts for confirmations each time you run something.
- The next thing you will need is a hostfile needed by MPI [see examples provided] Here is a sample hostfile:
  ```
  phi01.cselabs.umn.edu slots=16
  phi02.cselabs.umn.edu slots=16
  phi03.cselabs.umn.edu slots=16
  phi04.cselabs.umn.edu slots=16
  phi05.cselabs.umn.edu slots=16
  phi06.cselabs.umn.edu slots=16
  phi07.cselabs.umn.edu slots=16
  phi08.cselabs.umn.edu slots=16
  ```
- Call it `hostfile` or `phi_cluster` for example
- Run with, e.g.,
  ```
  mpirun -np 32 -hostfile phi_cluster test.ex
  ```