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:

  https://www.mcs.anl.gov/research/projects/mpi/index.html
- The LLNL link:

  https://computing.llnl.gov/tutorials/mpi/

Getting started: hello world from each process

```
#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 id (proc. 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 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 - a recent version of openmpi was installed (v. 4.0.3) – it is loaded by default.
- 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
- More information will be added as needed.

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

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

Asynchronus 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.

There should be an associated receive. General form:

```
MPI_Recv(start,count,datatyp,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.

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

MPI_TAG , MPI_SOURCE useful in case when MPI_ANY_TAG and /or MPI_ANY_SOURCE used in the receive

MPI_Get_count will tell us how much data of a particular type was received [details shortly]

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.

- Check on incoming messages (without rececevings them) with MPI_Iprobe or MPI_Probe

```
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 wait until a specific message arrives [with MPI_ANY_SOURCE]
- Can retrieve the length of a message, with MPI_Get_count :

```
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

```
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(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(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(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)

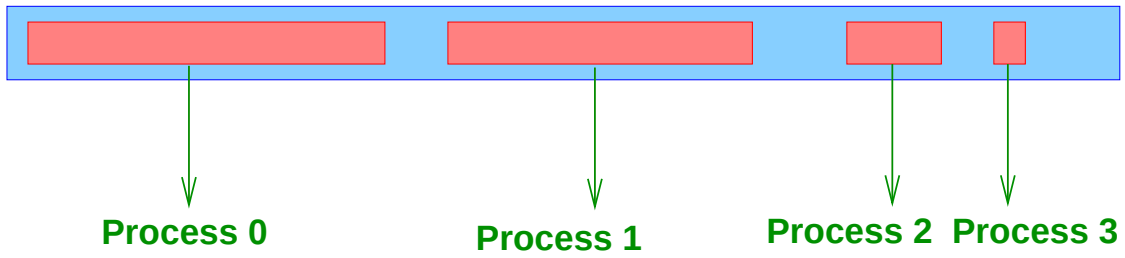
```
MPI_Gather (void *sendbuf,int sendcnt,
   MPI_Datatype sendtyp,void *recvbuf,int recvcnt,
   MPI_Datatype recvtyp,int root,MPI_Comm comm)
```

MPI Collective communication routines

Allgather	Allgatherv	Allreduce
Alltoall	Alltoallv	Bcast
Gather	Gatherv	Reduce
ReduceScatter	Scan	Scatter
Scatterv		

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

Scatterv:



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*
- Latest version of openMPI loaded by default (no need for modules)
- 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

Appendix: A few details and an example

- Some function calls

Send-Receive Pairs

```
#include <mpi.h>
int MPI_Send(const void *buf, int count,
    MPI_Datatype datatype, int dest,
    int tag, MPI_Comm comm)

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

Broadcast

```
#include <mpi.h>
int MPI_Bcast(void *buffer, int count,
    MPI_Datatype datatype, int root,
    MPI_Comm comm)
```

Quoting from the documentation: (open-mpi.org):

> *MPI_Bcast broadcasts a message from the process with rank root to all processes of the group, itself included. It is called by all members of group using the same arguments for comm, root. On return, the contents of root's communication buffer has been copied to all processes.*

- Main point: no need to have a corresponding receive - but all nodes in *comm* must issue the same command (the communication is "collective")
- What are MPI Data types?

MPI Datatypes

MPI_CHAR	MPI_C_COMPLEX
MPI_WCHAR	MPI_C_FLOAT_COMPLEX
MPI_SHORT	MPI_C_DOUBLE_COMPLEX
MPI_INT	MPI_C_LONG_DOUBLE_COMPLEX
MPI_LONG	MPI_C_BOOL
MPI_LONG_LONG_INT	MPI_LOGICAL
MPI_LONG_LONG	MPI_C_LONG_DOUBLE_COMPLEX
MPI_SIGNED_CHAR	MPI_INT8_T, MPI_INT16_T
MPI_UNSIGNED_CHAR	MPI_INT32_T
MPI_UNSIGNED_SHORT	MPI_INT64_T
MPI_UNSIGNED_LONG	MPI_UINT8_T, MPI_UINT16_T
MPI_UNSIGNED	MPI_UINT32_T
MPI_FLOAT	MPI_UINT64_T
MPI_DOUBLE	MPI_BYTE
MPI_LONG_DOUBLE	MPI_PACKED

Example: Compute values of e^x

- No practical value: Only goal is to illustrate a pipelined computation and sends, receives, broadcasts.

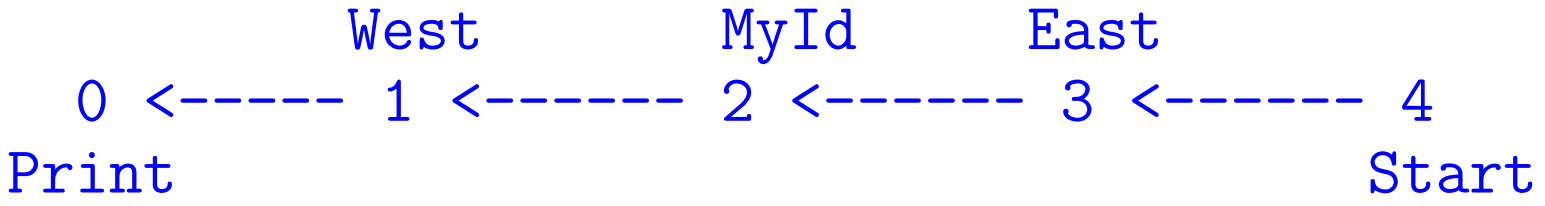
- We will compute $exp(x)$ by its n-th Taylor series expansion and use Horner's scheme:

$$e^x \approx 1 + \frac{x}{1}\left(1 + \frac{x}{2}\left(1 + \frac{x}{3}\left(\cdots\left(1 + \frac{x}{n}\right)\cdots\right)\right)\right)$$

- In Python this can be implemented as:

```
x = some-value
n = some value
val = 1.0
for i in range(n,0,-1):
    val = 1.0+(x/i)*val
print(val)
```

General procedure: Think of a linear array. For $n = 5$:



* Start: proc. $n - 1$ (last one) who sends
  value $val = 1 + (x/n)$ to West processor
* All other nodes: get val from East; update; and:
* Send val West *unless* MyId=0 in which case print.

- At first assume x is set in each process
- Next: Assume x is also sent along with val
- Next: Use a broadcast of x instead
- Next: In the above only one process is active at a time. What if I need to compute a bunch of values $e^{x_0}, e^{x_1}, \cdots e^{x_k}$

Example: Compute a sum of n numbers

Generate/ read n numbers, then scatter equal parts to each participating PE. Each PE computes a subsum then the whole sum is calculated by a reduction $(+)$ of subsums.

```
#include <mpi.h>
int MPI_Scatter(const void *sendbuf, int sendcnt,
    MPI_Datatype sendtype, void *recvbuf,
    int recvcount, MPI_Datatype recvtype,
    int root, MPI_Comm comm)
```

- Explore also the nonblocking send and scatter

```
#include <mpi.h>
int MPI_Isend(const void *buf, int count,
    MPI_Datatype datatype, int dest,int tag,
    MPI_Comm comm, MPI_Request *request)
```

```
int MPI_Iscatter(const void *sendbuf, int sendcnt,
    MPI_Datatype sendtype, void *recvbuf,
    int recvcount, MPI_Datatype recvtype,
    int root, MPI_Comm comm, MPI_Request *request)
```

➤ Recall: nonblocking sends allow you to do some work while a send is being processed. No need to wait until the send is completed. Work can be done as long as we do not touch the buffer being sent. Similarly for all non-blocking operations.

➤ Syntax for (regular) reduction operation

```
#include <mpi.h>
int MPI_Reduce(const void *sendbuf, void *recvbuf,
    int count, MPI_Datatype datatype, MPI_Op op,
    int root, MPI_Comm comm)
```

➤ *MPI_Ireduce*: adds a *MPI_Request *request* at end.

➤ Common *MPI_Op*'s:
MPI_SUM, MPI_PROD, MPI_MAX, MPI_MIN