CSCI 4850/5850 High-Performance Computing

Spring 2018

Tae-Hyuk (Ted) Ahn
Department of Computer Science
Program of Bioinformatics and Computational Biology
Saint Louis University
Learning Objectives

- Learn about point-to-point operation of MPI.
MPI Primer

- Initialization:
  
  ```
  int MPI_Init(int *argc, char ***argv)
  C++ example: MPI::Init(argc, argv);
  ```

- Get communicator size:
  
  ```
  int MPI_Comm_size(MPI_Comm comm, int *size)
  C++ example: int np= MPI::COMM_WORLD.Get_size();
  ```

- Get rank number:
  
  ```
  int MPI_Comm_rank(MPI_Comm comm, int *rank)
  C++ example: int rank= MPI::COMM_WORLD.Get_rank();
  ```

- Finalization:
  
  ```
  int MPI_Finalize(void)
  C++ example: MPI::Finalize();
  ```
Basic concepts: communicators

- MPI uses objects called communicators and groups to define which collection of processes may communicate with each other.
- Most MPI routines require you to specify a communicator as an argument.
- Communicators and groups will be covered in more detail later.
- For now, simply use `MPI_COMM_WORLD` (MPI::COMM_WORLD) whenever a communicator is required - it is the predefined communicator that includes all of your MPI processes.
Predefined MPI Constants

Important Predefined MPI Constants

- MPI::COMM_WORLD
- MPI::PROC_NULL
- MPI::ANY_SOURCE
- MPI::ANY_TAG
MPI - Hello World!

#include <iostream>
#include <cstdlib>              // has exit(), etc.
#include <mpi.h>               // MPI header file
#include <unistd.h>             // for sleep()

using namespace std;

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

    // initialize for MPI (should come before any other calls to
    //     MPI routines)
    MPI::Init(argc, argv);

    // get number of processes
    int nprocs = MPI::COMM_WORLD.Get_size();

    // get this process's number (ranges from 0 to nprocs - 1)
    int myid = MPI::COMM_WORLD.Get_rank();

    double starttime, endtime;
    if (myid == 0) {
        starttime = MPI::Wtime();
    }

    // print a greeting
    cout << "hello from process " << myid << " of " << nprocs << endl;
    sleep(5);

    if (myid == 0) {
        endtime = MPI::Wtime();
        cout << "Elapsed time = " << endtime-starttime << endl;
    }

    // clean up for MPI
    MPI::Finalize();

    return EXIT_SUCCESS;
}
MPI – How to build MPI binary?

[ahnt@apex Lab1]$ mpiCC
bash: mpiCC: command not found...
[ahnt@apex Lab1]$ which mpiCC
/usr/bin/which: no mpiCC in (/cm/shared/apps/slurm/15.08.6/sbin:/cm/shared/apps/slurm/15.08.6/bin:/cm/local/apps/gcc/5.2.0/bin:/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/cm/local/apps/environment-modules/3.2.10/bin:/home/ahnt/.local/bin:/home/ahnt/bin)

[ahnt@apex Lab1]$ module list
Currently Loaded Modulefiles:
  1) gcc/5.2.0      2) slurm/15.08.6

[ahnt@apex Lab1]$ module avail
[ahnt@apex Lab1]$ module add openmpi/gcc/64/1.10.1
[ahnt@apex Lab1]$ mpiCC -o mpihelloworld mpihelloworld.cpp
#include <iostream>
#include <cstdlib>              // has exit(), etc.
#include <mpi.h>                // MPI header file

using namespace std;

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

    // initialize for MPI (should come before any other calls to
    //     MPI routines)
    MPI::Init(argc, argv);

    // get number of processes
    int nprocs = MPI::COMM_WORLD.Get_size();

    // get this process's number (ranges from 0 to nprocs - 1)
    int myid = MPI::COMM_WORLD.Get_rank();

    // print a greeting
    cout << "hello from process " << myid << " of " << nprocs << endl;

    // clean up for MPI
    MPI::Finalize();

    return EXIT_SUCCESS;
Lab: Run mpi-helloworld at apex.slu.edu

- https://sites.google.com/a/slu.edu/atg/apex
- Prepare a script to submit the mpi helloworld program.
- Submit the job (sbatch) and check (squeue) the result.
- Useful SLURM commands:
  - squeue - lets you see what's in the SLURM queue
  - squeue -u <username> - as above but for a specific user
  - scancel <jobid> - cancels a certain job id
  - scancel -u <username> - cancels all jobs for a certain user
Point-to-point Operations

- The “Hello World” example did not contain any real communication.
- MPI Message: an array of elements of a particular MPI data type.
- MPI point-to-point communication routines generally have an argument list that takes one of the following formats:

<table>
<thead>
<tr>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocking sends</td>
<td>MPI_Send(buffer,count,type,dest,tag,comm)</td>
</tr>
<tr>
<td>Non-blocking sends</td>
<td>MPI_Isend(buffer,count,type,dest,tag,comm,request)</td>
</tr>
<tr>
<td>Blocking receive</td>
<td>MPI_Recv(buffer,count,type,source,tag,comm,status)</td>
</tr>
<tr>
<td>Non-blocking receive</td>
<td>MPI_Irecv(buffer,count,type,source,tag,comm,request)</td>
</tr>
</tbody>
</table>
Blocking vs. Non-blocking

- Most of the MPI point-to-point routines can be used in either blocking or non-blocking mode.

- **Blocking**:
  - A blocking send routine will only "return" after it is safe to modify the application buffer (your send data) for reuse. Safe means that modifications will not affect the data intended for the receive task. Safe does not imply that the data was actually received - it may very well be sitting in a system buffer.
  - A blocking send can be synchronous which means there is handshaking occurring with the receive task to confirm a safe send.
  - A blocking send can be asynchronous if a system buffer is used to hold the data for eventual delivery to the receive.
  - A blocking receive only "returns" after the data has arrived and is ready for use by the program.
Blocking vs. Non-blocking

- Most of the MPI point-to-point routines can be used in either blocking or non-blocking mode.

- **Non-blocking**:
  - Non-blocking send and receive routines behave similarly - they will return almost immediately. They do not wait for any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of message.
  - Non-blocking operations simply "request" the MPI library to perform the operation when it is able. The user can not predict when that will happen.
  - It is unsafe to modify the application buffer (your variable space) until you know for a fact the requested non-blocking operation was actually performed by the library. There are "wait" routines used to do this.
  - Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains.
Blocking vs. Non-blocking

- **Blocking** (MPI_Send() and MPI_Recv()) send routing will only return after it is safe to modify the buffer. “Safe” in this case means that modification will not affect the data to be sent. Safe does not imply that the data was actually received.

- **Non-blocking** (MPI_Isend() and MPI_Irecv()) send/receive routines return immediately. Non-blocking operations request that the MPI library perform the operation “when possible”. It is unsafe to modify the buffer until the requested operation has been performed. Non-blocking communications are primarily used to overlap computation with communication, optimizing performance.
Fairness

- MPI does not guarantee fairness - it's up to the programmer to prevent "operation starvation".
- Example: task 0 sends a message to task 2. However, task 1 sends a competing message that matches task 2's receive. Only one of the sends will complete.
Blocking Point-to-point Operations

- Synchronous instructions to send a message from one source rank to a destination rank:

  ```c
  int MPI_Send(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)
  ```

```c
i   MPI_Send(&data, count, type, j, tag, comm);
j   MPI_Recv(&data, count, type, i, tag, comm, &status);
```
MPI Basic (Blocking) Send

MPI_SEND(start, count, datatype, dest, tag, comm)

C++: void Comm::Send(const void* buf, int count, const Datatype& datatype, int dest, int tag) cons

- The message buffer is described by \((\text{start}, \text{count}, \text{datatype})\).
- The target process is specified by \(\text{dest}\), which is the rank of the target process in the communicator specified by \(\text{comm}\).
- When this function returns, the data has been delivered to the system and the buffer can be reused. The message may not have been received by the target process.
MPI Basic (Blocking) Receive

MPI_RECV(start, count, datatype, source, tag, comm, status)

C++: void Comm::Recv(void* buf, int count, const Datatype& datatype, int source, int tag, Status& status) const

- Waits until a matching (both source and tag) message is received from the system, and the buffer can be used
- source is rank in communicator specified by comm, or MPI_ANY_SOURCE (C++: MPI::ANY_SOURCE)
- tag is a tag to be matched on or MPI_ANY_TAG (C++: MPI::ANY_TAG)
- receiving fewer than count occurrences of datatype is OK, but receiving more is an error
- status contains further information (e.g. size of message)
Buffer

- Program (application) address space that references the data that is to be sent or received.
- In most cases, this is simply the variable name that is to be sent/received.
- For C or C++ programs, this argument is passed by reference and usually must be prepended with an ampersand: \texttt{&var1}.
Data Count

- Indicates the number of data elements of a particular type to be sent.
MPI Datatypes

- The data in a message to send or receive is described by a triple (address, count, datatype)

- An MPI datatype is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE)
  - a contiguous array of MPI datatypes
  - a strided block of datatypes
  - an indexed array of blocks of datatypes
  - an arbitrary structure of datatypes

- MPI functions to construct custom datatypes,
  - in particular ones for subarrays
<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>C Datatype</th>
<th>C++ Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI:CHAR</td>
<td>char</td>
<td>char</td>
</tr>
<tr>
<td>MPI:SHORT</td>
<td>signed short</td>
<td>signed short</td>
</tr>
<tr>
<td>MPI:INT</td>
<td>signed int</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI:LONG</td>
<td>signed long</td>
<td>signed long</td>
</tr>
<tr>
<td>MPI:LONG_LONG</td>
<td>signed long long</td>
<td>signed long long</td>
</tr>
<tr>
<td>MPI:SIGNED_CHAR</td>
<td>signed char</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI:UNSIGNED_CHAR</td>
<td>unsigned char</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI:UNSIGNED_SHORT</td>
<td>unsigned short</td>
<td>unsigned short</td>
</tr>
<tr>
<td>MPI:UNSIGNED</td>
<td>unsigned int</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI:UNSIGNED_LONG</td>
<td>unsigned long</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI:UNSIGNED_LONG_LONG</td>
<td>unsigned long long</td>
<td>unsigned long long</td>
</tr>
<tr>
<td>MPI:FLOAT</td>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>MPI:DOUBLE</td>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>MPI:LONG_DOUBLE</td>
<td>long double</td>
<td>long double</td>
</tr>
<tr>
<td>MPI:BOOL</td>
<td>bool</td>
<td>Complex&lt;float&gt;</td>
</tr>
<tr>
<td>MPI:COMPLEX</td>
<td></td>
<td>Complex&lt;double&gt;</td>
</tr>
<tr>
<td>MPI:DOUBLE_COMPLEX</td>
<td></td>
<td>Complex&lt;long double&gt;</td>
</tr>
<tr>
<td>MPI:LONG_DOUBLE_COMPLEX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI:WCHAR</td>
<td>wchar_t</td>
<td>wchar_t</td>
</tr>
<tr>
<td>MPI:BYTE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI:PACKED</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Destination**

- An argument to send routines that indicates the process where a message should be delivered.
- Specified as the rank of the receiving process.
Source

- An argument to receive routines that indicates the originating process of the message.
- Specified as the rank of the sending process.
- This may be set to the wild card `MPI_ANY_SOURCE` (C++: `MPI::ANY_SOURCE`) to receive a message from any task.
Tag

- Arbitrary non-negative integer assigned by the programmer to uniquely identify a message.
- Send and receive operations should match message tags.
- For a receive operation, the wild card `MPI_ANY_TAG` (C++ `MPI::ANY_TAG`) can be used to receive any message regardless of its tag.
- The MPI standard guarantees that integers 0-32767 can be used as tags, but most implementations allow a much larger range than this.
Communicator

- Indicates the communication context, or set of processes for which the source or destination fields are valid.
- Unless the programmer is explicitly creating new communicators, the predefined communicator `MPI_COMM_WORLD (MPI::COMM_WORLD)` is usually used.
Status

- The source or tag of a received message may not be known if wildcard values were used in the receive operation.

- Also, if multiple requests are completed by a single MPI function, a distinct error code may need to be returned for each request.

- The information is returned by the status argument of MPI_RECV.

- The status argument also returns information on the length of the message received.

- However, this information is not directly available as a field of the status variable and a call to MPI_GET_COUNT is required to "decode" this information.
MPi Basic Send/Receive

- We need to fill in the details in

\[
\begin{align*}
\text{Process 0} & \quad \text{Send}(\text{data}) & \quad \text{Process 1} \\
\quad \quad & \quad \quad & \quad \quad \\
\quad & \quad \quad & \quad \quad \\
\quad & \quad \quad & \quad \quad \\
\quad & \quad \quad & \quad \quad \\
\text{Receive}(\text{data})
\end{align*}
\]

- Things that need specifying:
  - How will “data” be described?
  - How will processes be identified?
  - How will the receiver recognize/screen messages?
  - What will it mean for these operations to complete?
Drawbacks of Blocking non-buffered send/recv

- Idling overheads
  - Be posted at roughly simultaneously
  - Asynchronous environment, hard to predict

- Deadlocks
  - Break the cycle waits
  - Inverse the sequence, but more buggy

\[
\begin{align*}
\text{P0} & \quad \begin{array}{l}
\text{send}(\&a, 1, 1); \\
\text{receive}(\&b, 1, 1);
\end{array} \\
\text{P1} & \quad \begin{array}{l}
\text{send}(\&a, 1, 0); \\
\text{receive}(\&b, 1, 0);
\end{array}
\end{align*}
\]
Buffers

When you send data, where does it go? One possibility is:

Process 0
- User data
  - Local buffer
  - the network

Process 1
- Local buffer
  - User data
Buffers

Typically, a system buffer area is reserved to hold data in transit. For example:

Path of a message buffered at the receiving process
Blocking operation

- So far we have been using *blocking* communication:
  - `MPI_Recv` does not complete until the buffer is full (available for use).
  - `MPI_Send` does not complete until the buffer is empty (available for use).

- Completion depends on size of message and amount of system buffering.
#include <iostream>
#include <cstdlib>              // has exit(), etc.
#include <mpi.h>                // MPI header file

using namespace std;

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

    MPI::Status status;

    // initialize for MPI (should come before any other calls to
    //   MPI routines)
    MPI::Init(argc, argv);

    // get number of processes
    int nprocs = MPI::COMM_WORLD.Get_size();

    // get this process's number (ranges from 0 to nprocs - 1)
    int myid = MPI::COMM_WORLD.Get_rank();

    // send message from rank 0
    int val;
    if (myid == 0) {
        val = 100;
        MPI::COMM_WORLD.Send(&val, 1, MPI::INT, 1, 0);
    }

    // rank 1 receive message from rank 0
    else if (myid == 1) {
        MPI::COMM_WORLD.Recv(&val, 1, MPI::INT, 0, 0, status);
        cout << "Process 1 received number " << val << "from process 0\n" << endl;
    }

    // clean up for MPI
    MPI::Finalize();

    return EXIT_SUCCESS;
}