PHY1610 - Distributed Parallel Programming with MPI

Ramses van Zon

March 29, 2022

Improving scalability

Issues with shared memory programming

- Parallel tasks are run by threads.
- All threads live on the same node and share the memory.
- Limited to the resources of a single node.
- Creation and deletion of threads can cause overhead (see assignment 8!)
- Can lead to bugs like race conditions.

Improving scalability

Issues with shared memory programming

- Parallel tasks are run by threads.
- All threads live on the same node and share the memory.
- Limited to the resources of a single node.
- Creation and deletion of threads can cause overhead (see assignment 8!)
- Can lead to bugs like race conditions.

Today will look at distributed memory programming

- Parallel tasks are processes.
- Each process has only its own, private memory.
- Processes need not be on the same node.
- You can scale up the size of your system to as many resources as you have.
- Harder to create race condition bugs, but now you get new bugs like dead-lock.
- Must explicitly code in the communication between processes: Message Passing Interface aka MPI

Section 1

MPI Intro

Message Passing Interface (MPI)

What is it?

- An open standard library interface for message passing, ratified by the MPI Forum
- Version: 1.0 (1994), 1.1 (1995), 1.2 (1997), 1.3 (2008)
- Version: 2.0 (1997), 2.1 (2008), 2.2 (2009)
- Version: 3.0 (2012), 3.1 (2015)
- Version: 4.0 (2021)

Message Passing Interface (MPI)

What is it?

- An open standard library interface for message passing, ratified by the MPI Forum
- Version: 1.0 (1994), 1.1 (1995), 1.2 (1997), 1.3 (2008)
- Version: 2.0 (1997), 2.1 (2008), 2.2 (2009)
- Version: 3.0 (2012), 3.1 (2015)
- Version: 4.0 (2021)

MPI Implementations

- OpenMPI www.open-mpi.org

  ```
  $ module load gcc/9 openmpi/4
  ```

  or

  ```
  $ module load intel openmpi
  ```

  Currently these give you OpenMPI version 4.1.2..

- MPICH www.mpich.org (MPICH, MVAPICH2, IntelMPI)

  ```
  $ module load intel intelmpi
  ```

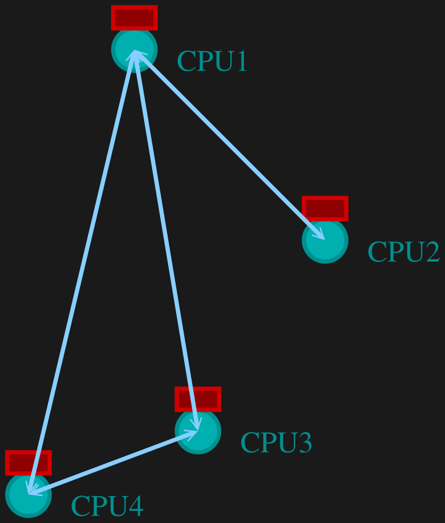
MPI is a Library for Message-Passing

Library:

- Not built in to compiler.
- Function calls that can be made from any compiler, many languages.
- Just link to it.
- Wrappers: mpicc, mpif90, mpicxx

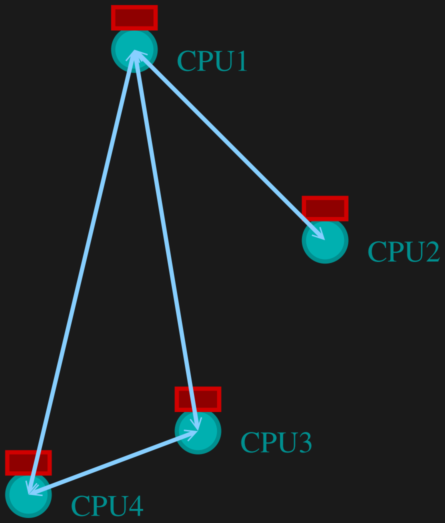
```
#include <iostream>
#include <string>
#include <mpi.h>
using namespace std;

int main(int argc, char **argv)
{
    int rank, size;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    cout << "Hello from task " +
            to_string(rank) + " of " +
            to_string(size) + "\n";

    MPI_Finalize();
}
```

MPI is a Library for Message Passing



- Communication/coordination between tasks done by sending and receiving messages.
- Each message involves a function call from each of the programs.

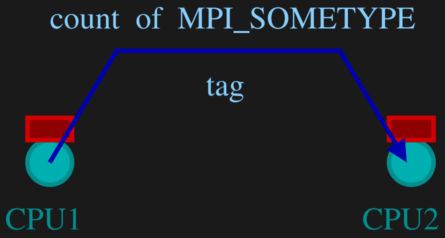
MPI is a Library for Message Passing



Three basic sets of functionality:

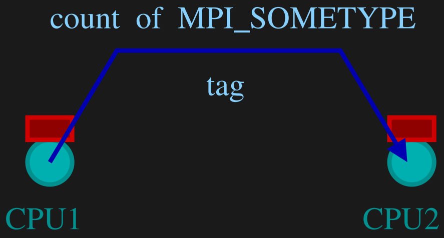
- Pairwise communications via messages;
- Collective operations via messages;
- Efficient routines for getting data from memory into messages and vice versa.

Messages



- Messages have a **sender** and a **receiver**.
- When you are sending a message, you don't need to specify the sender (it is the current processor).
- A sent message has to be actively received by the receiving process

Messages



- MPI messages are a string of length **count** all of some fixed MPI **type**.
- MPI types exist for characters, integers, floating point numbers, etc.
- An arbitrary non-negative integer **tag** is also included – helps keep things straight if lots of messages are sent.

Size of MPI Library

- Many, many functions (>200).
- Not nearly so many concepts.
- We'll get started with just 10-12, use more as needed.

```
MPI_Init()
MPI_Comm_size()
MPI_Comm_rank()
MPI_Ssend()
MPI_Recv()
MPI_Finalize()
```

Example: Hello World

```
#include <iostream>
#include <string>
#include <mpi.h>
using namespace std;

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<< "Hello from task" + to_string(rank) +
          " of " + to_string(size) + " world\n";

   MPI_Finalize();
}
```

Example: Hello World

Compile with MPI

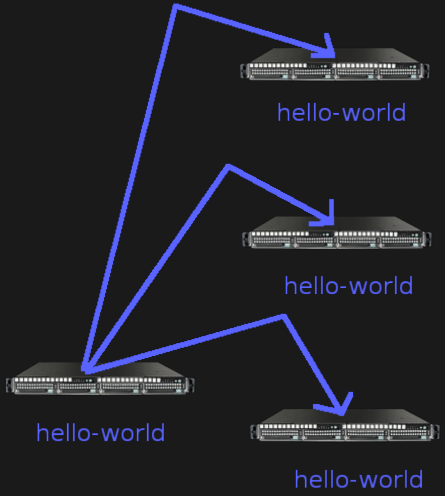
MPI provides compiler wrappers

- mpicc
- mpicxx
- mpif90

that set all the -I, -L, -l, etc. options properly for the base compiler.

```
$ git clone /scinet/course/phy1610/mpi
$ cd mpi
$ module load gcc/9 openmpi/4
$ mpicxx -O2 -std=c++17 -o mpi-hello-world mpi-hello-world.cc  # or: 'make mpi-hello-world'
$ mpirun -n 16 ./mpi-hello-world
```

What `mpirun` Does



- Launches *n* processes, assigns each an MPI **rank** and starts the program.
- Usually, the processes run the same executable, therefore **each process runs the exact same code**.
- For multinode runs, has a list of nodes, and logs in (effectively) to each node, where it launches the program.
- Most MPI implementations have a more versatile but non-portable `mpirun` command as well.

Number of Processes

- Number of processes to use is almost always equal to the number of processors.
- But not necessarily.
- On a Teach debugjob, what happens when you run this?

```
$ mpirun -n 16 ./mpi-hello-world
```

mpirun runs *any* program

- mpirun will start its process-launching procedure for any program.
- Sets variables somehow that mpi programs recognize so that they know which process they are.

E.g., try this:

```
$ hostname
$ mpirun -n 4 hostname
$ ls
$ mpirun -n 4 ls
```

Example: Hello World

```
$ mpirun -n 4 ./mpi-hello-world
Hello from task 2 of 4 world
Hello from task 1 of 4 world
Hello from task 0 of 4 world
Hello from task 3 of 4 world
```

```
$ mpirun --tag-output -n 4 ./mpi-hello-world
[1,2]<stdout>:Hello from task 2 of 4
[1,3]<stdout>:Hello from task 3 of 4
[1,0]<stdout>:Hello from task 0 of 4
[1,1]<stdout>:Hello from task 1 of 4
```

The `--tag-output` flag is specific for the OpenMPI implementation of MPI.

Section 2

MPI Basics

MPI Basics

Basic MPI Components

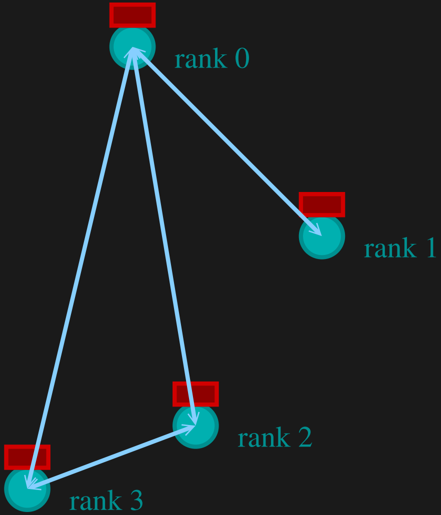
- `#include <mpi.h>`
  MPI library definitions
- `MPI_Init(&argc,&argv)`
  MPI Intialization, must come first
- `MPI_Finalize()`
  Finalizes MPI, must come last
- Formally, MPI routines return an error code. But in fact, MPI applications by default abort when there is an error.

Communicator Components

- A communicator is a handle to a group of processes that can communicate.
- `MPI_Comm_rank(MPI_COMM_WORLD,&rank)`
- `MPI_Comm_size(MPI_COMM_WORLD,&rank)`

```
#include <iostream>
#include <string>
#include <mpi.h>
using namespace std;

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 << "Hello from task" + to_string(rank) +
            " of " + to_string(size) + " world\n";

    MPI_Finalize();
}
```

Communicators

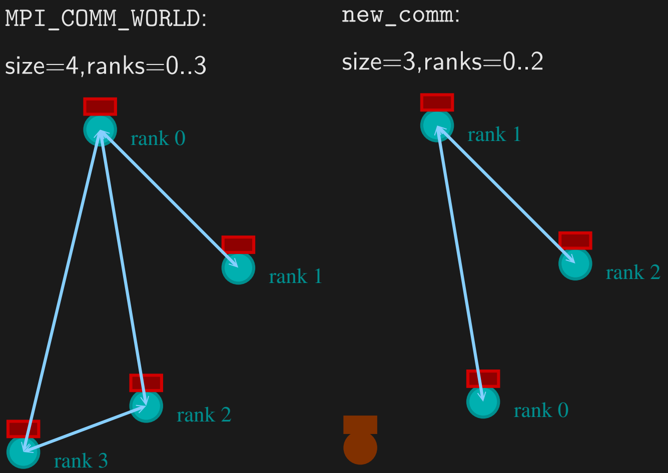


- MPI groups processes into communicators.
- Each communicator has some size – number of tasks.
- Every task has a rank 0..size-1
- Every task in your program belongs to `MPI_COMM_WORLD`.

`MPI_COMM_WORLD`:
size = 4, ranks = 0..3

Communicators

- One can create one's own communicators over the same tasks.
- May break the tasks up into subgroups.
- May just re-order them for some reason.



MPI Basics - Communicator Components

- `MPI_COMM_WORLD`:

  Global Communicator

- `MPI_Comm_rank(MPI_COMM_WORLD,&rank)`

  Get current tasks rank

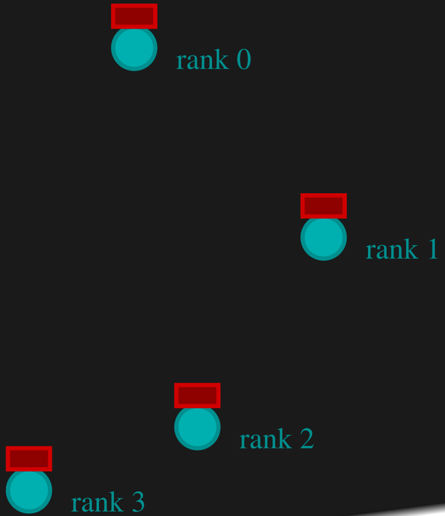
- `MPI_Comm_size(MPI_COMM_WORLD,&size)`

  Get communicator size

MPI = Rank and Size

Rank and Size are much more important in MPI than in OpenMP

- In OpenMP, the compiler assigns jobs to each thread; you do not need to know which one is which (usually).
- In MPI, all proceses run the same code.
- In MPI, processes determine amongst themselves which piece of puzzle to work on, based on their **rank**, then communicate with appropriate others.

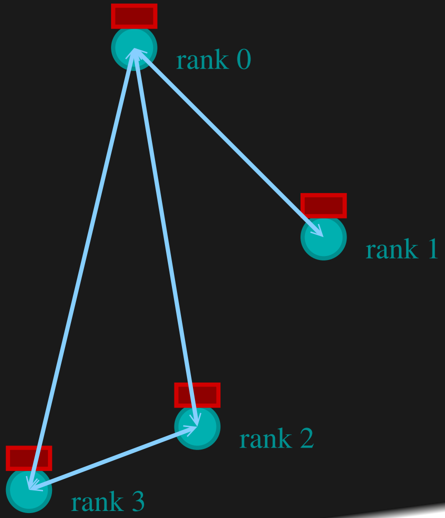


MPI = Communication

Explicit Communication between Tasks

- In OpenMP, threads can communicate using the memory.
- In MPI, a process which needs data of another process needs to communicate with that process by passing messages.

```
MPI_Ssend(...)

MPI_Recv(...)
```



MPI: Send & Receive

```
MPI_Ssend(sendptr, count, MPI_TYPE, destination,tag, Communicator);

MPI_Recv(recvptr, count, MPI_TYPE, source, tag, Communicator, MPI_status)
```

- `sendptr/recvptr`: pointer to message
- `count`: number of elements in message
- MPI_TYPE: one of MPI_DOUBLE, MPI_FLOAT, MPI_INT, MPI_CHAR, etc.
- `destination/source`: rank of sender/reciever
- `tag`: unique id for message pair
- `Communicator`: MPI_COMM_WORLD or user created
- `status`: receiver status (error, source, tag)

Note: MPI has a Fortran and C interface. We can use the C interface in C++ but will have to deal with pointers, i.e., we'll give arguments likes &(array[0]) or array.data() instead of just array.

MPI: Send & Receive

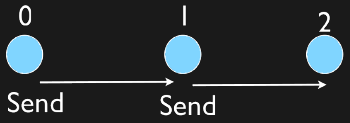
```
#include <iostream>
#include <string>
#include <mpi.h>
using namespace std;
int main(int argc, char **argv)
{
    int rank, size;
    int tag = 1;
    double msgsent, msgrcvd;
    MPI_Status rstatus;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    msgsent = 111.;
    msgrcvd = -999.;
    if (rank == 0) {
        MPI_Ssend(&msgsent, 1, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD);
        cout << "Sent " + to_string(msgsent) + " from " + to_string(rank) + "\n";
    }
    if (rank == 1) {
        MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD, &rstatus);
        cout << "Received " + to_string(msgrcvd) + " on " + to_string(rank) + "\n";
    }
    MPI_Finalize();
}
```

MPI: Send & Receive

```
$ make firstmessage
$ mpirun -n 2 ./firstmessage
Send 111.000000 from 0
Received 111.000000 on 1
```

MPI Communication Patterns

Send a message to the right:



Specials

Special Source/Destination MPI_PROC_NULL

MPI_PROC_NULL basically ignores the relevant operation; can lead to cleaner code.

Special Source MPI_ANY_SOURCE

MPI_ANY_SOURCE is a wildcard; matches any source when receiving.

Special Status MPI_STATUS_IGNORE

Use MPI_STATUS_IGNORE if you do not want to capture the status in a receive.

Section 3

Deadlocks

Deadlocks are a classic parallel bug

- In this explicit message passing model, it is possible to completely freeze the application.
- This can happen when a process is sending a message, but no process is or will ever be ready to receive it.
- This is called **deadlock**
- To see how that could happen, let's look at an example.

MPI: Send Right, Receive Left

```
#include <iostream>
#include <string>
#include <mpi.h>
using namespace std;
int main(int argc, char **argv)
{
    int         rank, size, left, right, tag = 1;
    double      msgsent, msgrcvd;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    left = rank - 1;
    if (left < 0) left = MPI_PROC_NULL;
    right = rank + 1;
    if (right >= size) right = MPI_PROC_NULL;
    msgsent = rank*rank;
    msgrcvd = -999.;
    MPI_Ssend(&msgsent, 1, MPI_DOUBLE, right, tag, MPI_COMM_WORLD);
    MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, left, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    cout << to_string(rank) + ": Sent " + to_string(msgsent)
          + " and got " + to_string(msgrcvd) + "\n";
    MPI_Finalize();
}
```

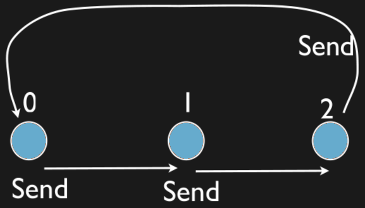
MPI: Send Right, Receive Left

```
$ make secondmessage
$ mpirun -n 3 ./secondmessage
2: Sent 4.000000 and got 1.000000
0: Sent 0.000000 and got -999.000000
1: Sent 1.000000 and got 0.000000
$
```

```
$ mpirun -n 6 ./secondmessage
4: Sent 16.000000 and got 9.000000
5: Sent 25.000000 and got 16.000000
0: Sent 0.000000 and got -999.000000
1: Sent 1.000000 and got 0.000000
2: Sent 4.000000 and got 1.000000
3: Sent 9.000000 and got 4.000000
```

MPI: Send Right, Receive Left with Periodic BCs

Periodic Boundary Conditions:



MPI: Send Right, Receive Left with Periodic BCs

```
    ...
    left = rank - 1;
    if (left < 0) left = size-1; // Periodic BC
    right = rank + 1;
    if (right >= size) right =0; // Periodic BC
    msgsent = rank*rank;
    msgrcvd = -999.;
    ...
```

MPI: Send Right, Receive Left with Periodic BCs

```
    ...
    left = rank - 1;
    if (left < 0) left = size-1; // Periodic BC
    right = rank + 1;
    if (right >= size) right =0; // Periodic BC
    msgsent = rank*rank;
    msgrcvd = -999.;
    ...
```

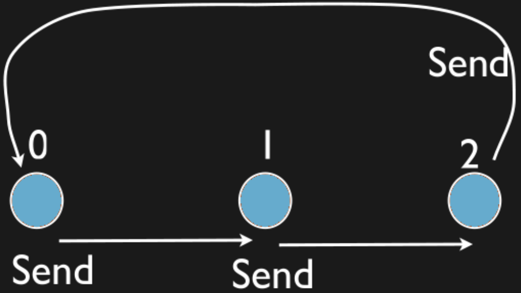
```
$ make thirdmessage
$ mpirun -n 3 ./thirdmessage

_
```

Program hangs!

Deadlock!

- A classic parallel bug.
- Occurs when a cycle of tasks are waiting for the others to finish.
- Whenever you see a closed cycle, you likely have (or risk) a deadlock.
- Here, all processes are waiting for the send to complete, but no one is receiving.

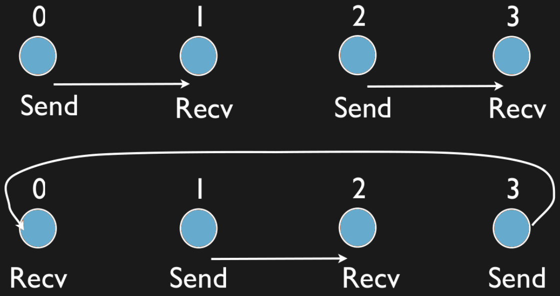
Sends and receives must be paired when sending



How do we fix the deadlock?

Without using new MPI routine, how do we fix the deadlock?

Even-odd solution



- First: evens send, odds receive
- Then: odds send, evens receive
- Will this work with an odd number of processes? How about 2? 1?

MPI: Send Right, Recv Left with Periodic BCs - fixed

```
    ...
    if ((rank % 2) == 0) {
        MPI_Ssend(&msgsent, 1, MPI_DOUBLE, right, tag, MPI_COMM_WORLD);
        MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, left, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    } else {
        MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, left, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Ssend(&msgsent, 1, MPI_DOUBLE, right, tag, MPI_COMM_WORLD);
    }
    ...
```

```
$ make fourthmessage
$ mpirun -n 5 ./fourthmessage
1: Sent 1.000000 and got 0.000000
2: Sent 4.000000 and got 1.000000
3: Sent 9.000000 and got 4.000000
4: Sent 16.000000 and got 9.000000
0: Sent 0.000000 and got 16.000000
```

MPI: Sendrecv

```
MPI_Sendrecv(sendptr, count, MPI_TYPE, destination, tag,
             recvptr, count, MPI_TYPE, source, tag, Communicator, MPI_Status)
```

- A blocking send and receive built together.
- Lets them happen simultaneously.
- Can automatically pair send/recvs.
- Why 2 sets of tags/types/counts?

Send Right, Receive Left with Periodic BCs - Sendrecv

Code

```
    ...
    MPI_Sendrecv(&msgsent, 1, MPI_DOUBLE, right, tag,
                 &msgrcvd, 1, MPI_DOUBLE, left, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    ...
```

Execution

```
$ make fifthmessage
$ mpirun -n 5 ./fifthmessage
1: Sent 1.000000 and got 0.000000
2: Sent 4.000000 and got 1.000000
3: Sent 9.000000 and got 4.000000
4: Sent 16.000000 and got 9.000000
0: Sent 0.000000 and got 16.000000
```