Introduction to High Performance Computing

Alejandro Cárdenas-Avendaño

KONRAD LORENZ
FUNDACIÓN UNIVERSITARIA

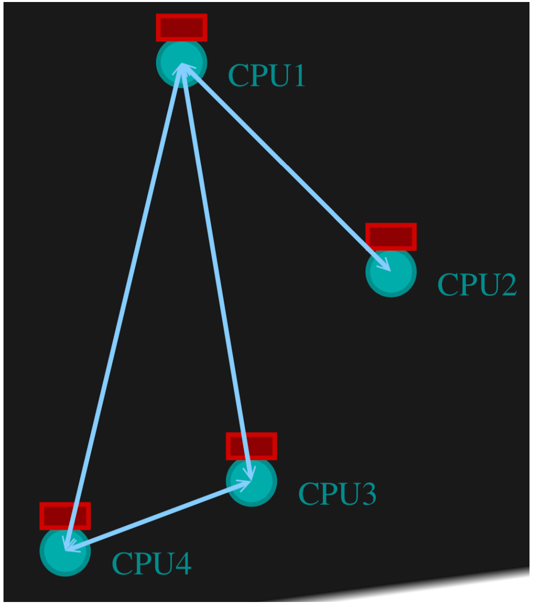
MPI Intro

Message Passing Interface (MPI)

- An open standard library interface for message passing
- OpenMPI www.open-mpi.org
- 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

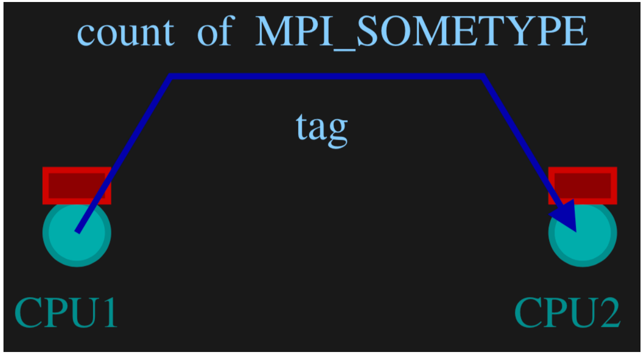
MPI is a Library for Message Passing

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

```cpp
#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();

return 0; }
```

```
$ mpicxx -o Hello HelloWorld.cpp
$ mpiexec -np 2 ./Hello
Hello from task 1 of 2
Hello from task 0 of 2
```

What *mpirun/mpiexec* Does

- Launches n processes, assigns each an MPI rank and starts the program.
- 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.

```
$ mpic++ -o Hello HelloWorld.cpp
$ mpirun -np 2 ./Hello
Hello from task 1 of 2
Hello from task 0 of 2
```

mpiexec runs any program

- mpiexec 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
$ mpiexec -n 2 hostname
$ ls
$ mpiexec -n 2 ls
```

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

mpiexec runs any program

- mpiexec 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
$ mpiexec -n 2 hostname
$ ls
$ mpiexec -n 2 ls
```

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

```
$ mpiexec --tag-output -n 2 ./Hello
[1,0]<stdout>:Hello from task 0 of 2
[1,1]<stdout>:Hello from task 1 of 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

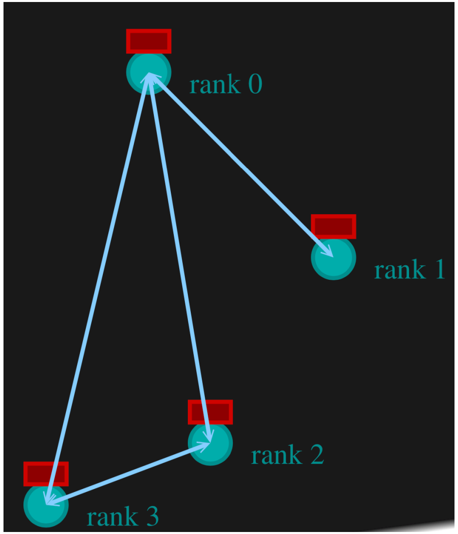
- Formally, MPI routines return an error code. But in fact, MPI applications by default abort when there is an error.

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,&size)

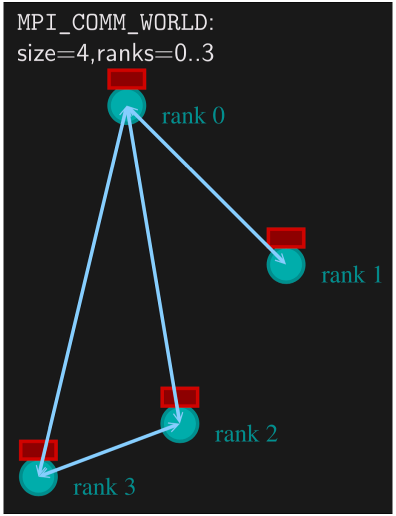
Communicators

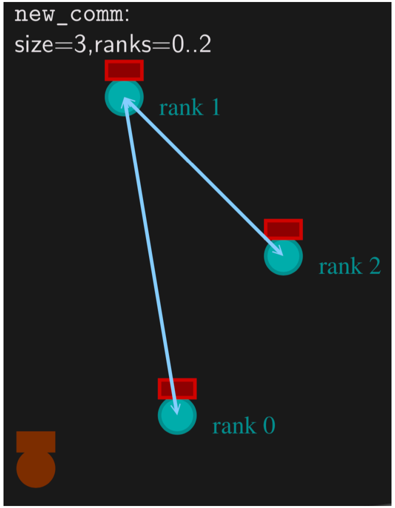
- MPI groups processes into communicators.
- Each communicator has some size – number of tasks.
- Every task has a rank 0..size-1
- Every task in your program belongs to MPI_COMM_WORLD.



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)

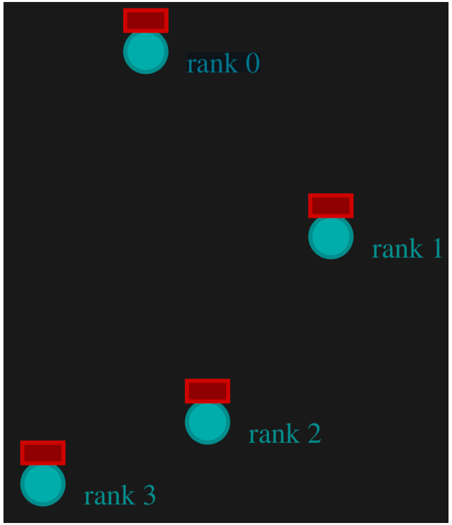
Get current tasks rank

- MPI_Comm_size(MPI_COMM_WORLD,&size)

Get communicator size

MPI = Rank and Size

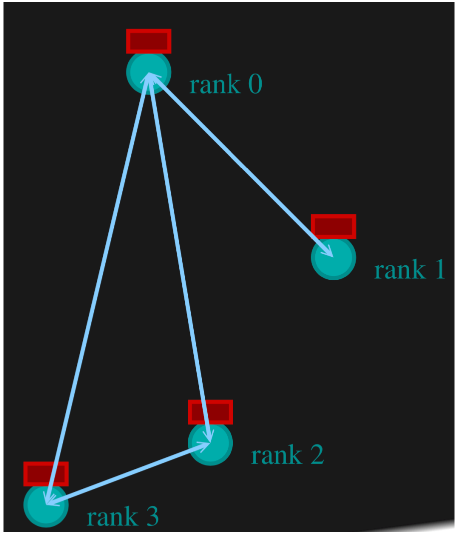
- Rank and Size are much more important in MPI than in OpenMP
- In OpenMP, the compiler assigns jobs to each thread; you do not need to know which one is which (usually).
- In MPI, processes determine amongst themselves which piece of puzzle to work on, then communicate with appropriate others.



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)

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 = 137.;
    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();
return 0;
}
```

```
$ mpicxx -o Comm1 firstmessage.cpp
$ mpiexec -np 2 ./Comm1
Received 137.000000 on 1
Sent 137.000000 from 0
```

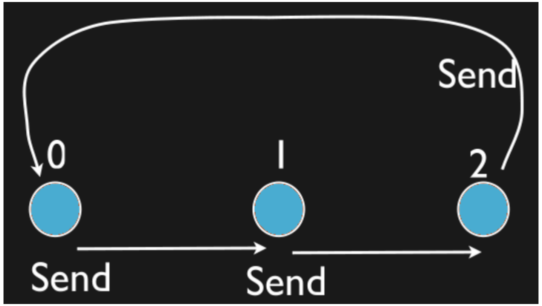
MPI: Send Right, Receive Left

```
#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_Status   rstatus;
    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, &rstatus);
    cout << to_string(rank) + ": Sent " + to_string(msgsent) + " and got " +
to_string(msgrcvd) + "\n";
    MPI_Finalize();
 }
```

```
$ mpicxx -o Comm2 secondmessage.cpp
$ mpiexec -np 2 ./Comm2
1: Sent 1.000000 and got 0.000000
0: Sent 0.000000 and got -999.000000
```

MPI: Send Right, Receive Left with Periodic BCs

- Periodic Boundary Conditions:




```
...
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.;
...
```

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.

All sends and receives must be paired at the time of sending

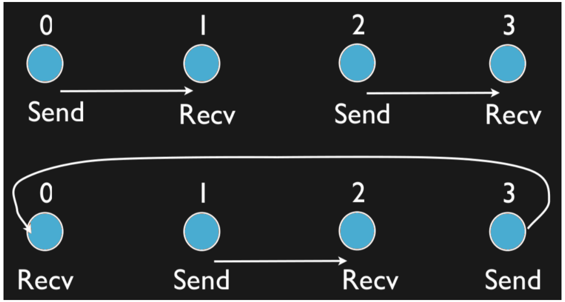
Deadlocks

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

How do we fix the deadlock?

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



- 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, &rstatus);
} else {
    MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, left, tag, MPI_COMM_WORLD, &rstatus);
    MPI_Ssend(&msgsent, 1, MPI_DOUBLE, right, tag, MPI_COMM_WORLD);
}
...
```

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.
- Let them happen simultaneously.
- Can automatically pair send/recvs.
- Why 2 sets of tags/types/counts?

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

```
...
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, &rstatus);
} else {
    MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, left, tag, MPI_COMM_WORLD, &rstatus);
    MPI_Ssend(&msgsent, 1, MPI_DOUBLE, right, tag, MPI_COMM_WORLD);
}
...
```

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