High Performance Scientific Computing
MPI I.

SciNet HPC Consortium
University of Toronto

Winter 2014

Lecture 21

- MPI Library
- mpirun, mpic++
- MPI Basics
- Send/Recv
- Sendrecv

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)

MPI Implementations

- OpenMPI (www.open-mpi.org)
  - OpenMPI 1.4.x, 1.6.x
  - SciNet GPC: **`module load gcc openmpi`**
  - SciNet GPC: **`module load intel openmpi`**
- MPICH2 (www.mpich.org)
  - MPICH2, MVAPICH2, IntelMPI
  - SciNet GPC: **`module load intel intelmpi`**

MPI is a **Library** for Message-Passing

- Not built in to compiler
- Function calls that can be made from any compiler, many languages
- Just link to it
- Wrappers: `mpicc`, `mpif77`

C

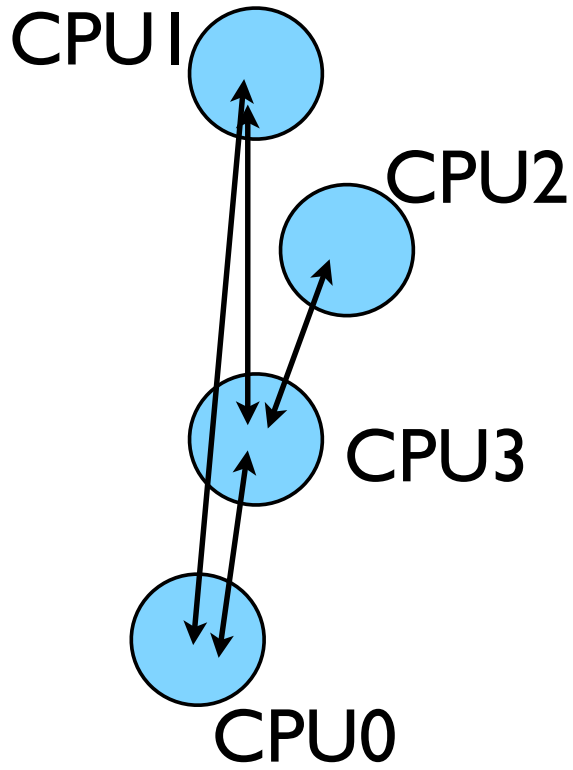
```c
#include <stdio.h>
#include <mpi.h>

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

        int rank, size;
        int ierr;

        ierr = MPI_Init(&argc, &argv);

        ierr = MPI_Comm_size(MPI_COMM_WORLD, &size);
        ierr = MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        printf("Hello from task %d of %d, world!\n", rank, size);

        MPI_Finalize();

        return 0;
}
```

Fortran

```fortran
program hellompiworld
include "mpif.h"

integer rank, size
integer ierr

call MPI_INIT(ierr)

call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)

print *, "Hello from task ", rank, " of ", size, ", world!"

call MPI_FINALIZE(ierr)

return
end
```

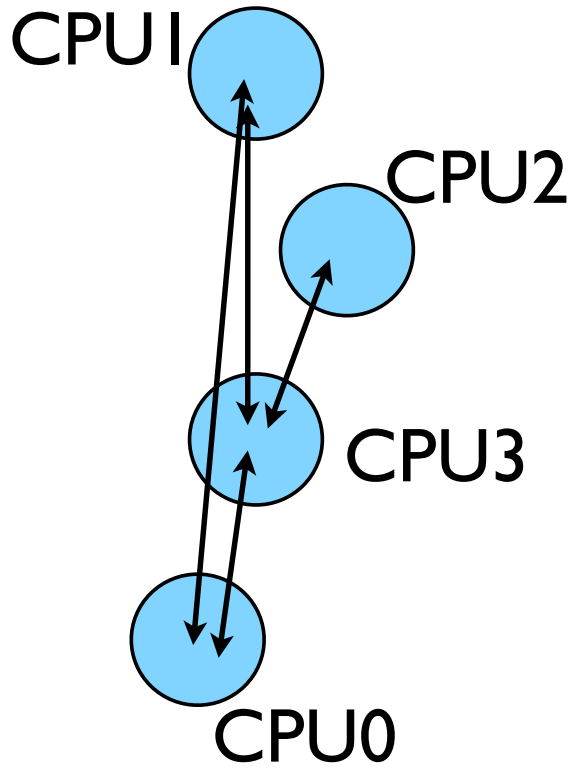
MPI is a Library for **Message-Passing**

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



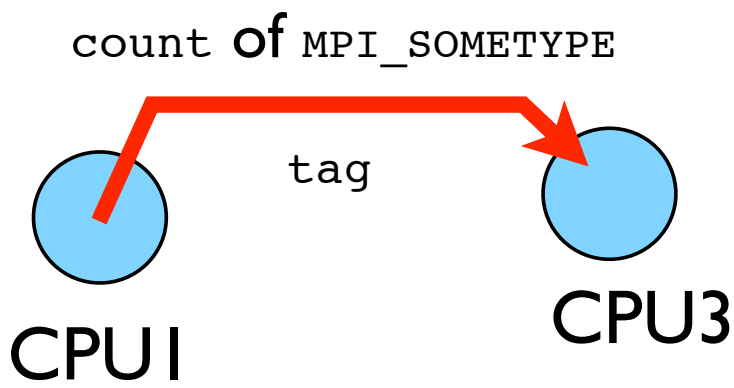
MPI is a Library for **Message-Passing**

- 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



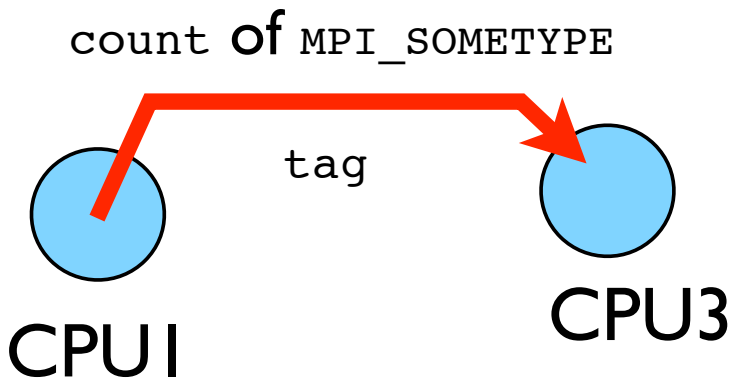
Messages

- Messages have a **sender** and a **receiver**
- When you are sending a message, don't need to specify sender (it's 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 <mpi.h>
int main(int argc, char **argv) {
  int rank, size;
  int ierr;
  ierr = MPI_Init(&argc, &argv);

  ierr = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  ierr = MPI_Comm_size(MPI_COMM_WORLD, &size);

  std::cout<<''Hello from task ''<<rank
  <<'' of ''<<size<<'' world'';

  ierr = MPI_Finalize();

  return 0;
}
```

Example: "Hello World"

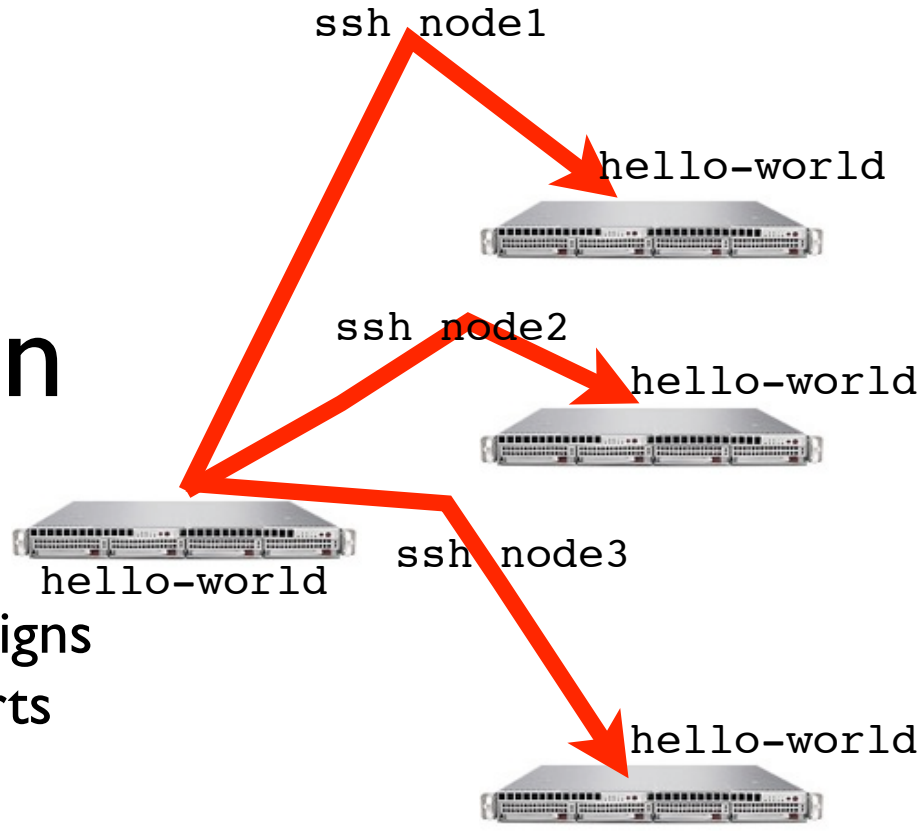
Compile with MPI

- Provides compiler wrappers (**mpicc, mpic++, mpif77, ...**) that sets all the -I, -L, libs, etc. properly for the base compiler

```
$module load gcc/4.8.1 openmpi/gcc/1.6.4
$mpic++ -o mpihello mpi_hello_world.cc
```

What mpirun does

- Launches n processes, assigns each an MPI rank and starts the program
- For multinode run, has a list of nodes, ssh's to each node and launches the program



Number of Processes

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

```
$ mpirun -np 24 hello-world
```

mpirun runs *any* program

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

```
$ hostname
$ mpirun -np 4 hostname
$ ls
$ mpirun -np 4 ls
```

Example: "Hello World"

```
$mpirun -np 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
```

Example: "Hello World"

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

Example: "Hello World"

```
#include <iostream>
#include <mpi.h>
int main(int argc, char **argv) {
  int rank, size;
  int ierr;
  ierr = MPI_Init(&argc, &argv);

  ierr = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  ierr = MPI_Comm_size(MPI_COMM_WORLD, &size);

  std::cout<<''Hello from task ''<<rank
  <<'' of ''<<size<<'' world'';

  ierr = MPI_Finalize();

  return 0;
}
```

MPI Basics

Basic MPI Components

- `#include <mpi.h>` : MPI library details
- `MPI_Init(&argc, &argv);` : MPI Intialization, must come first
- `MPI_Finalize()` : Finializes MPI, must come last
- `ierr` : Returns error code

MPI Basics

Basic MPI Components

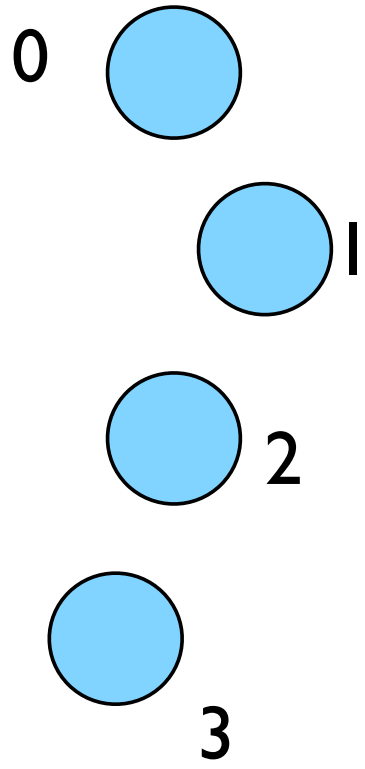
- `#include <mpi.h>` : MPI library details
- `MPI_Init(&argc, &argv);` : MPI Intialization, must come first
- `MPI_Finalize()` : Finializes MPI, must come last
- `ierr` : Returns error code

Communicator Components

- `MPI_Comm_rank(MPI_COMM_WORLD, &rank)`
- `MPI_Comm_size(MPI_COMM_WORLD, &size)`

Communicators

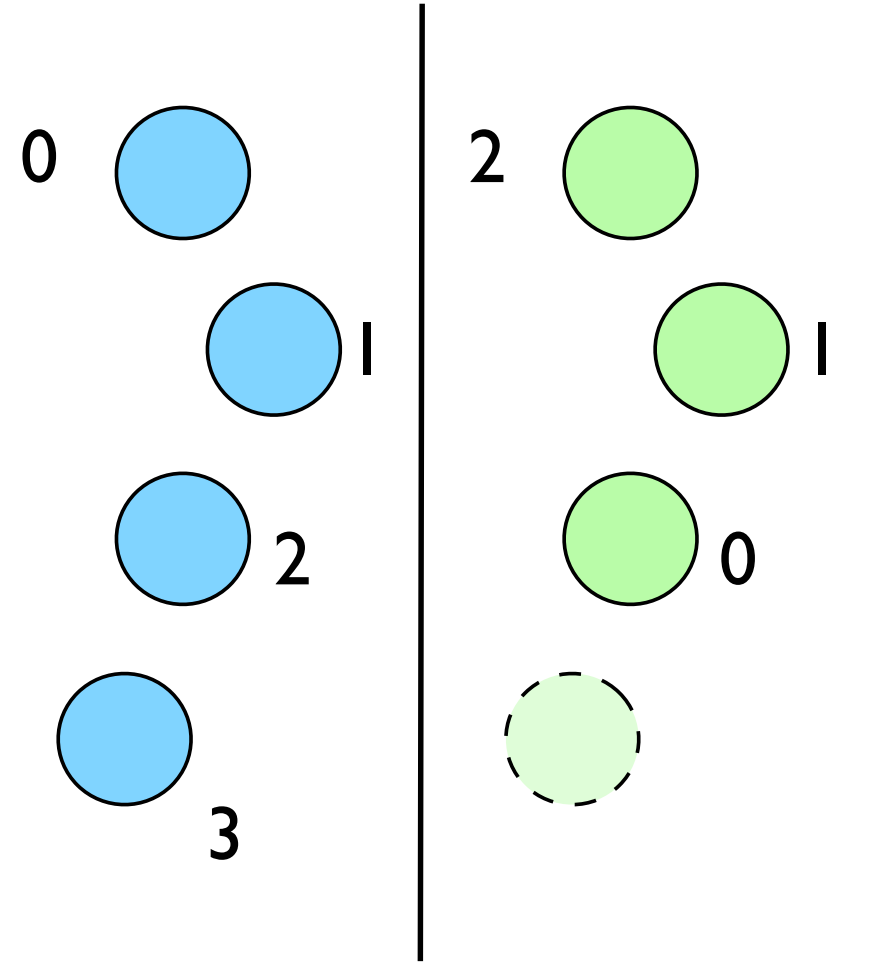
- MPI groups processes into communicators.
- Each communicator has some size -- number of tasks.
- Each task has a rank 0..size-1
- Every task in your program belongs to `MPI_COMM_WORLD`



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

Communicators

- Can create our own communicators over the same tasks
- May break the tasks up into subgroups
- May just re-order them for some reason



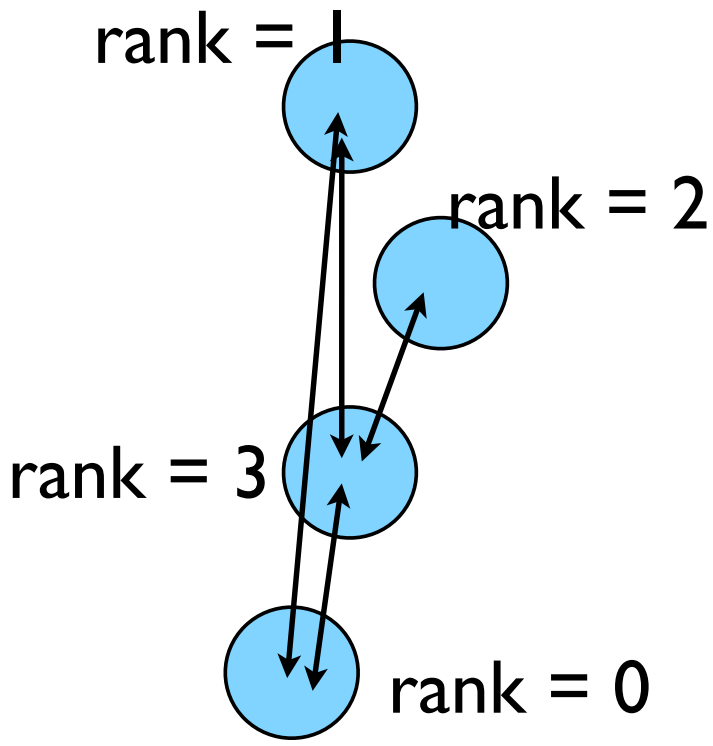
MPI Basics

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

Rank and Size much more important in MPI than OpenMP

- In OpenMP, compiler assigns jobs to each thread; don't need to know which one you are.
- MPI: processes determine amongst themselves which piece of puzzle to work on, then communicate with appropriate others.



MPI: Send & Receive

```
ierr = MPI_Ssend(sendptr, count, MPI_TYPE,
destination,tag, Communicator)
ierr = MPI_Recv(rcvptr, count, MPI_TYPE,
source, tag,Communicator, MPI_status)
```

- **sendptr/rcvptr**: pointer to message
- **count**: number of elements in ptr
- **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 <mpi.h>
using namespace std;
int main(int argc, char **argv) {
  int rank, size, ierr;
  int tag=1;
  double msgsent, msgrcvd;
  MPI_Status rstatus;
  ierr = MPI_Init(&argc, &argv);
  ierr = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  ierr = MPI_Comm_size(MPI_COMM_WORLD, &size);
  //Set Messages
  msgsent = 111.;
  msgrcvd = -999.;
  ...
}
```

MPI: Send & Receive

```
{
  ...
  if ( rank == 0) {
    ierr = MPI_Ssend(&msgsent, 1, MPI_DOUBLE, 1,
    tag, MPI_COMM_WORLD);
    cout<<"\n Send "<<msgsent<<" from "<<rank;
  }
  if ( rank == 1) {
    ierr = MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, 0, tag,
    MPI_COMM_WORLD, &rstatus);
    cout<<"\n Recieved "<<msgrcvd<<" on "<<rank;
  }
  ierr = MPI_Finalize();

  return 0;
}
```

MPI: Send & Receive

```
$mpirun -np 2 ./firstmessage
Send 111 from 0
Recieved 111 on 1
```

Special Source/Dest: MPI_PROC_NULL

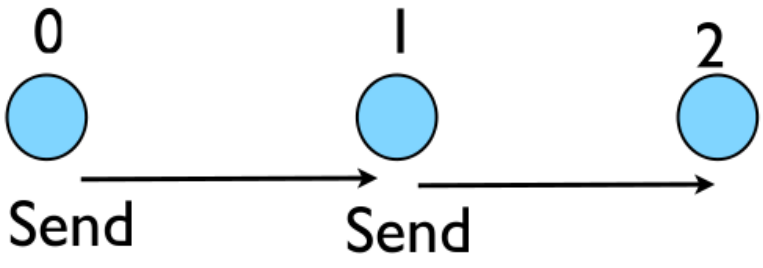
`MPI_PROC_NULL` basically ignores the relevant operation; can lead to cleaner code.

Special Source: MPI_ANY_SOURCE

`MPI_ANY_SOURCE` is a wildcard; matches any source when receiving.

MPI: Send Left, Receive Right

Send Message to the Left



MPI: Send Left, Receive Right

```
#include <iostream>
#include <mpi.h>
using namespace std;
int main(int argc, char **argv) {
  int rank, size, ierr;
  int left, right;
  int tag=1;
  double msgsent, msgrcvd;
  MPI_Status rstatus;
  ierr = MPI_Init(&argc, &argv);
  ierr = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  ierr = MPI_Comm_size(MPI_COMM_WORLD, &size);
  ...
}
```

MPI: Send Left, Receive Right

```
{
  //Pass to left
  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.;

  ierr = MPI_Ssend(&msgsent, 1, MPI_DOUBLE, right,
  tag, MPI_COMM_WORLD);
  ierr = MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, left,
  tag, MPI_COMM_WORLD, &rstatus);
  cout<<rank<<":  Sent "<<msgsent<<" and got
  "<<msgrcvd<<endl;
  ierr = MPI_Finalize();
  return 0;
}
```

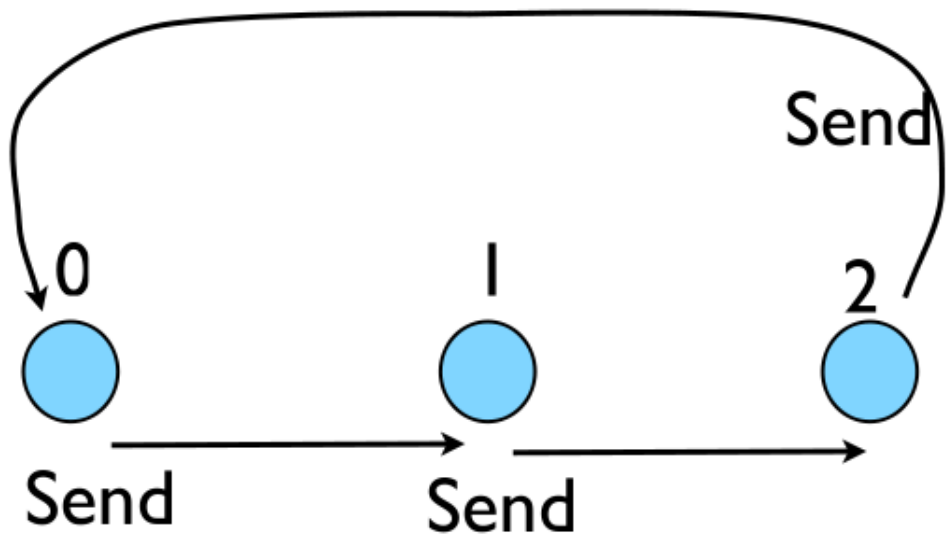
MPI: Send Left, Receive Right

```
$mpirun -np 3 ./secondmessage
2:  Sent 4 and got 1
0:  Sent 0 and got -999
1:  Sent 1 and got 0
```

```
$mpirun -np 6 ./secondmessage
4:  Sent 16 and got 9
5:  Sent 25 and got 16
0:  Sent 0 and got -999
1:  Sent 1 and got 0
2:  Sent 4 and got 1
3:  Sent 9 and got 4
```

MPI: Send Left, Receive Right with Periodic BC's

Periodic BC's

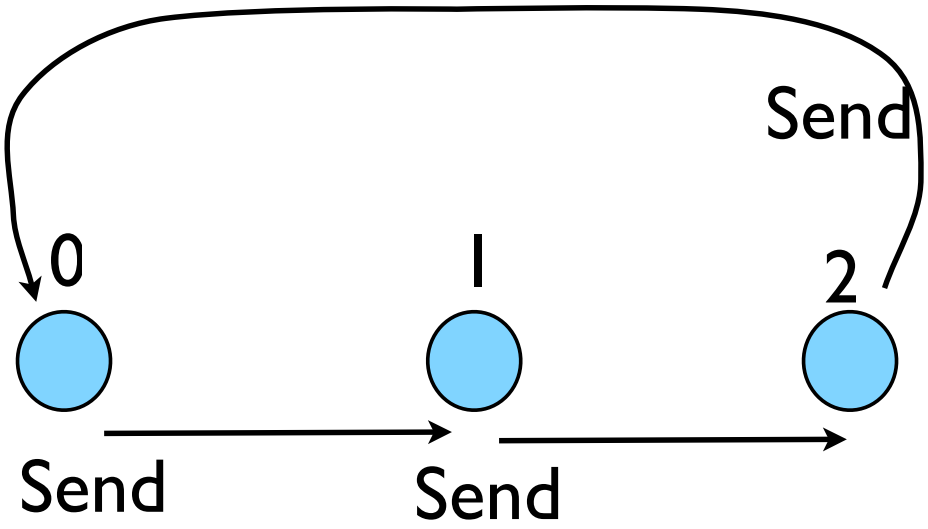


MPI: Send Left, Receive Right with Periodic BC's

```
{
  ...
  //Pass to left
  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.;
  ...
}
```

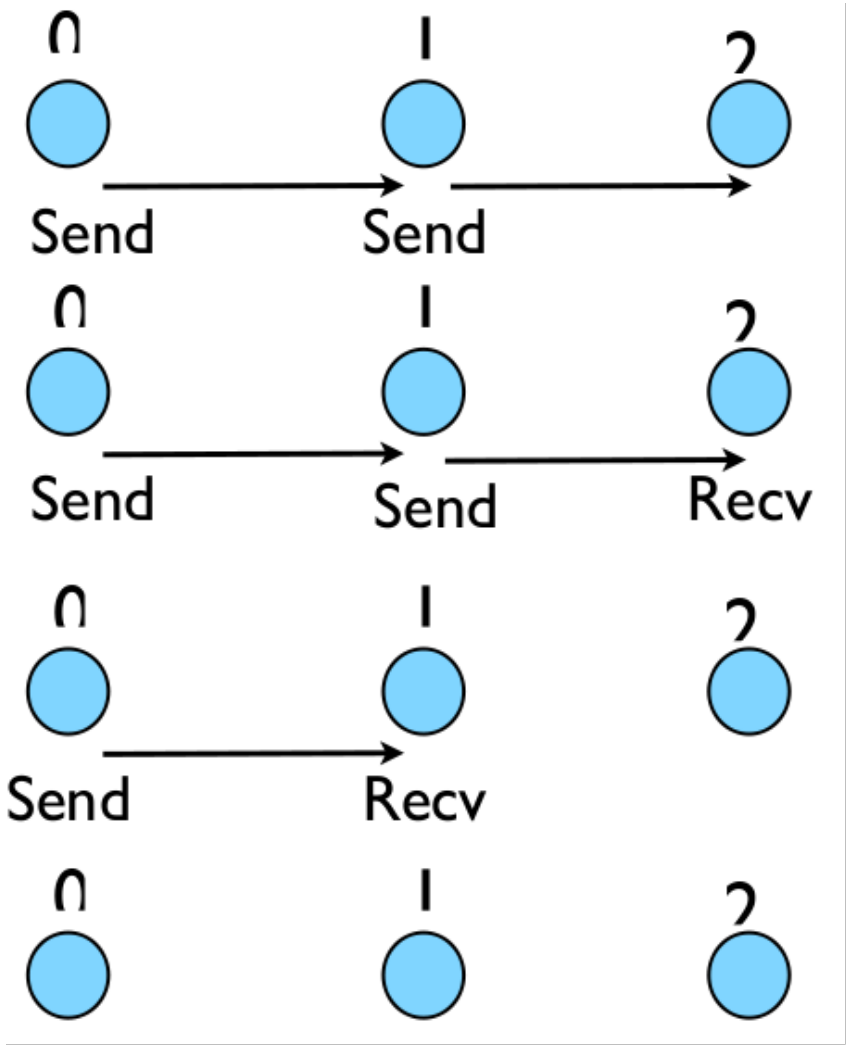
Deadlock

- A classic parallel bug
- Occurs when a cycle of tasks are for the others to finish.
- Whenever you see a closed cycle, you likely have (or risk) deadlock.



MPI: Send Left, Receive Right with Periodic BC's

Message Progression



Big MPI
Lesson #1

All sends and receives must be paired, **at time of sending**