Designing Parallel Program

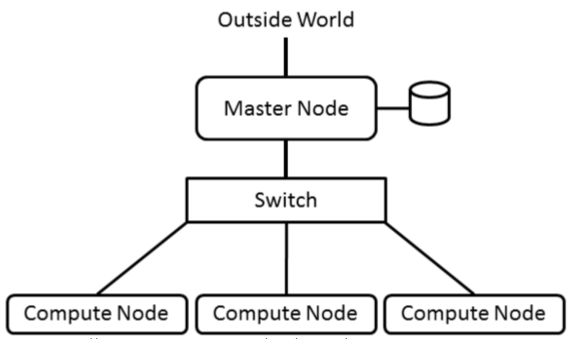
Tutorial

Lec 3

Agenda

- Introduction to cluster computers
- Introduction to Message Passing Interface
- Point to Point Communication

Computing Cluster



http://www.admin-magazine.com/HPC/Articles/Building-an-HPC-Cluster

Basic Sun Grid Engine User commands

- Submitting Batch Jobs to SGE
- Monitoring SGE Jobs

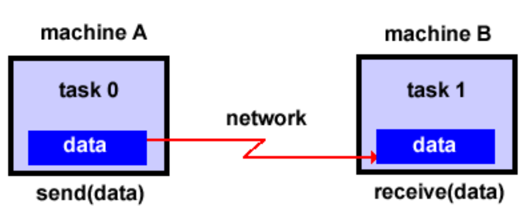
Chapter 3. Using:
http://www.rocksclusters.org/roll-documentation/sge/4.2.1/roll-sge-usersguide.pdf

Agenda

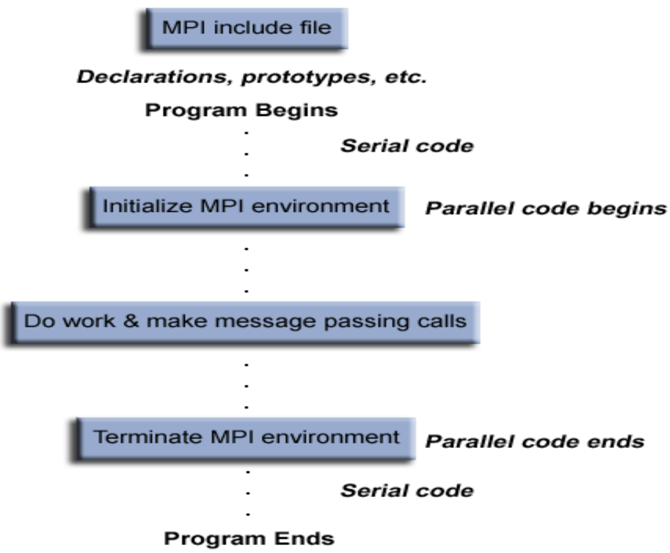
- Introduction to Message Passing Interface
- Point to Point Communication

Message Passing Model

- Message passing model allows processors to communicate by passing messages:
  - Typically processors do not share memory
- Data transfer between processors required cooperative operations to be performed by each processor:
  - One processor *sends* the message while other *receives* the message

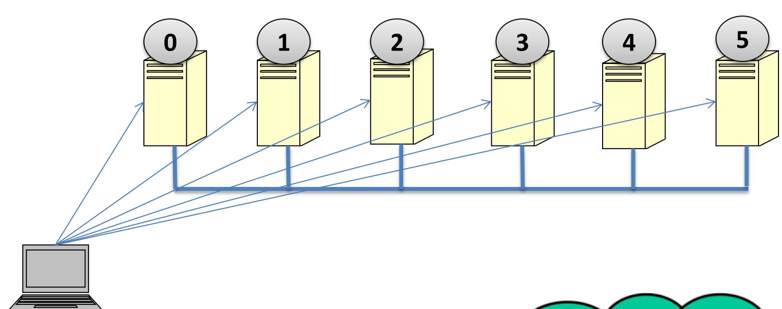


General MPI program structure



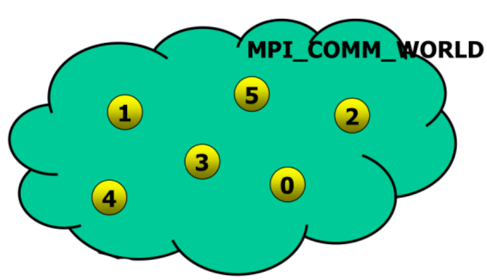
7

MPI Execution Model



Rank varies from ***0-5***

Size is ***6***



SPMD Programming

- *Single Program Multiple Data* in which all participant processors (or processes) run the same program image, but operate on their local memory contents.

- A special case of the more general *MIMD* (*Multiple Instruction Multiple Data*) model, in which different participants may run different local programs.
  - e.g. a common paradigm in *pre-MPI* days was for one node to run a *host* program that coordinated I/O and controlled the other nodes, which ran a separate *worker* programs that did most of the computation.

9

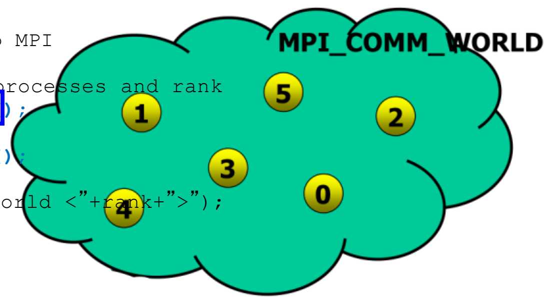
Hello World in Java

```
import java.util.*;
import mpi.*;

public class HelloWorld {

   public static void main(String args[]) throws Exception {

   // Initialize MPI
   MPI.Init(args); // start up MPI

      // Get total number of processes and rank
   size = MPI.COMM_WORLD.Size();
   rank = MPI.COMM_WORLD.Rank();

   System.out.println("Hello World <"+rank+">");

   MPI.Finalize();

      }
}
```

10

After Initialization

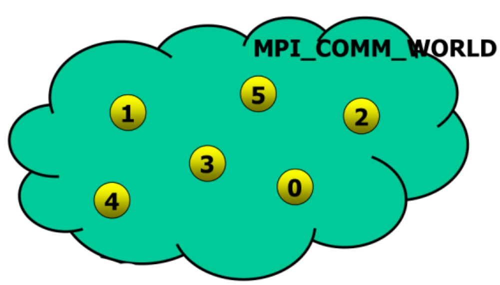
```
import java.util.*;
import mpi.*;

public class HelloWorld {

   public static void main(String args[]) throws Exception {

   // Initialize MPI
   MPI.Init(args); // start up MPI

      // Get total number of processes and rank
   size = MPI.COMM_WORLD.Size();

   rank = MPI.COMM_WORLD.Rank();

   System.out.println("Hello World <"+rank+">");

   MPI.Finalize();

     }
}
```



- *Total number of processes in a communicator:*
  - *The size of MPI.COMM_WORLD is 6*

What is rank?

```
..
   // Get ID of current process
     rank = MPI.COMM_WORLD.Rank();

..
```



- *The "unique" identify (id) of a process in a communicator:*
  - *Each of the six processes in MPI.COMM_WORLD has a distinct rank or id*

Communicators

- Defines the scope of a communication operation.
  Processes have **ranks** associated with the communicator.
- Initially, all processes enrolled in a "universe" called **MPI.COMM_WORLD**, and each process is given a unique rank, a number from 0 to p - 1, with p processes.
- Other communicators can be established for groups of processes.
  A set of MPI routines exists for forming communicators.

Installing Java MPI library

- Two examples:
  - MPJ Express
  - OpenMPI with java bindings

MPJ Express

- Java 1.6 (stable) or higher (Mandatory).
- Download MPJ Express and unpack it
- Set MPJ_HOME and PATH variables
  - `export MPJ_HOME=/path/to/mpj/`
  - `export PATH=$MPJ_HOME/bin:$PATH`
- Complie MPJ Express program:
  - javac -cp .:$MPJ_HOME/lib/mpj.jar HelloWorld.java

http://mpjexpress.org/guides.html

MPJ Express

- MPJ Express provides four different ways of executing the MPI program. We will only look into two modes.
  - Multicore mode: `mpjrun.sh -np 2 HelloWorld`
  - Cluster mode:
    - Create machinefile:
      machine1
      machine2
    - Start daemons: mpjboot machines
    - mpjrun.sh -np 2 -dev niodev HelloWorld
    - mpjhalt machines

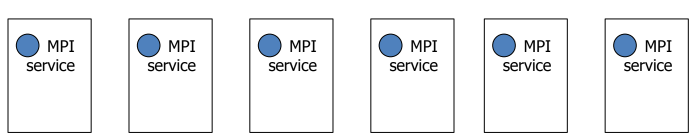
Openmpi with java binding

- Download Openmpi and unpack it
- Installing Openmpi with java enable.
  - `./configure --enable-mpi-java`
  - `make`
  - `make install`
- Compile: `mpijavac Hello.java`
- Execute: `mpirun java Hello`

https://www.open-mpi.org/faq/?category=java

Scenario for Running an MPI Program

- Pool of available host computers, each running an *MPI service* or *MPI daemon*.

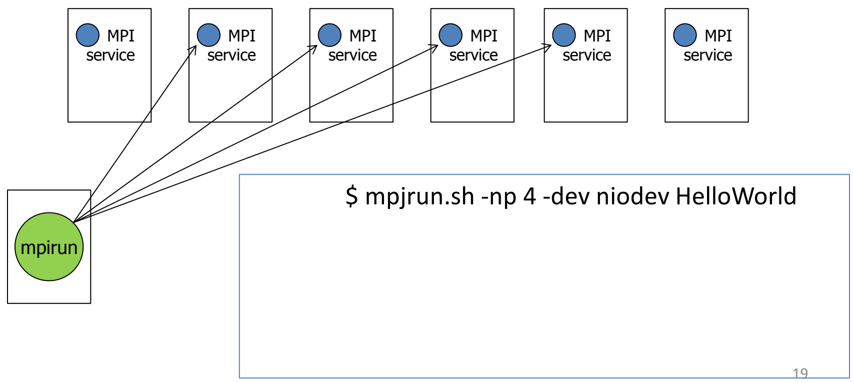


$ mpjboot.sh machines

Content of machines file
compute-0-1
compute-0-2
compute-0-3
compute-0-4
compute-0-5
compute-0-6

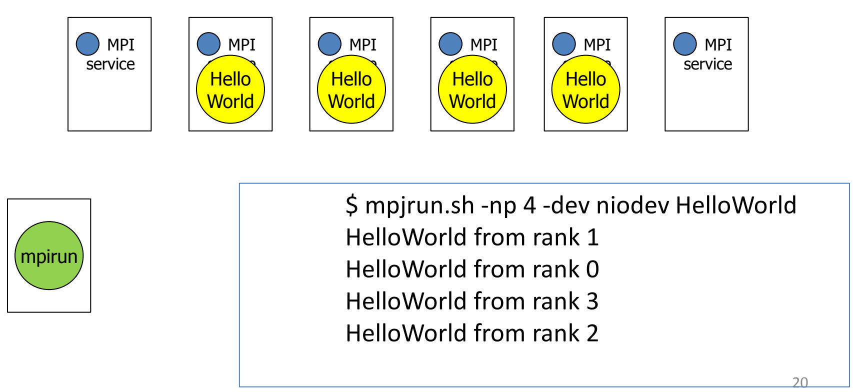
Scenario for Running an MPI Program

- Client program, e.g. mpirun, connects to *P* daemons and asks them to start processes.



Scenario for Running an MPI Program

- *Hello World* process (say) starts on *P* hosts.

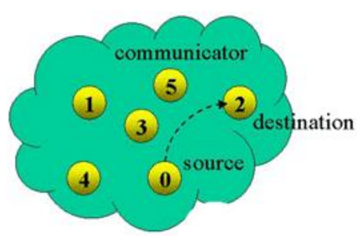


Agenda

- Introduction to Message Passing Interface
- Point to Point Communication

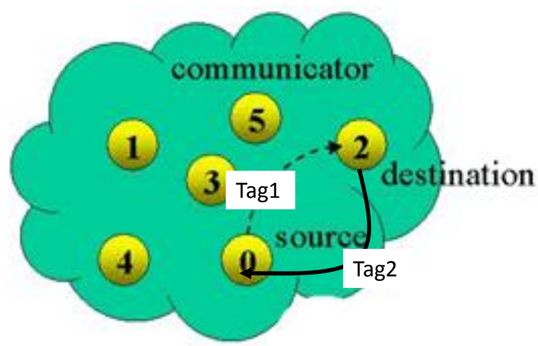
Point to Point Communication

- The most fundamental facility provided by MPI
- Basically "exchange messages between two processes":
  - One process (*source*) sends message
  - The other process (destination) receives message

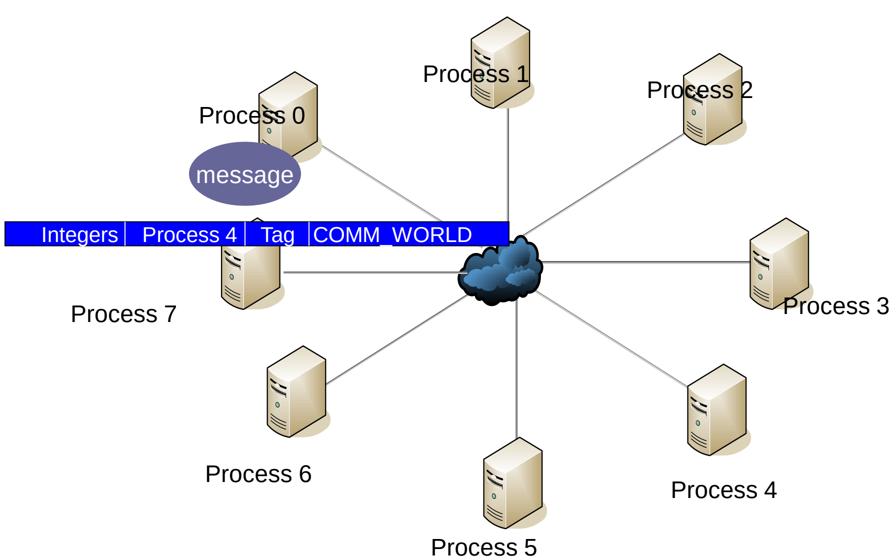


Point to Point Communication

- It is possible to send message for each basic datatype:
  - Floats, Integers, Doubles …
- Each message contains a “tag”—an identifier

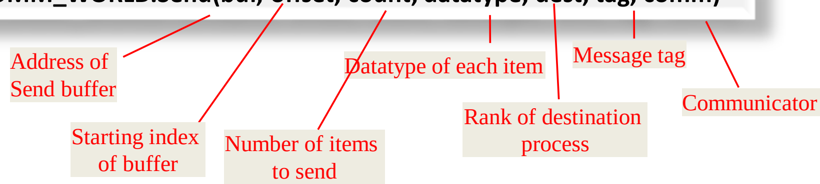


Point to Point Communication



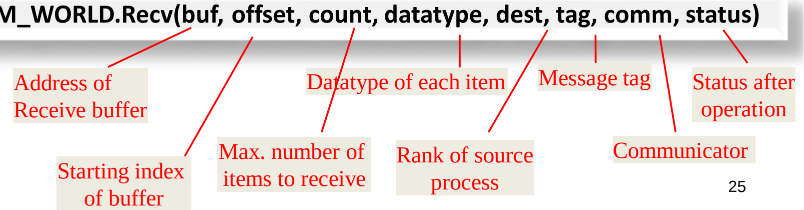
Blocking Send

```
MPI.COMM_WORLD.Send(buf, offset, count, datatype, dest, tag, comm)
```

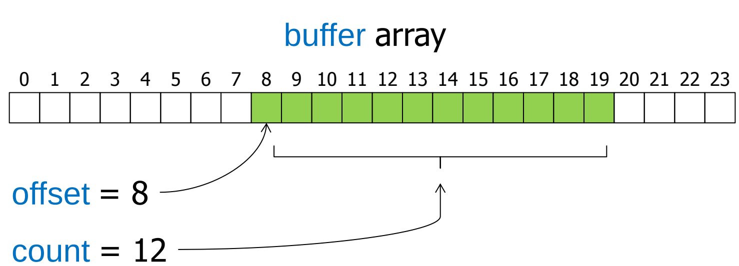


Blocking Receive

```
MPI.COMM_WORLD.Recv(buf, offset, count, datatype, dest, tag, comm, status)
```



Buffer Example

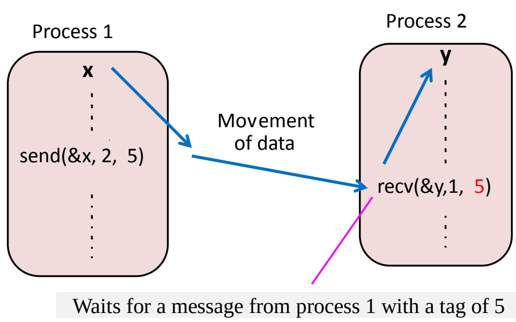


- Elements of buffer actually sent are in green.
- Note offset is often 0.
- count may take value 1 to send a single element.

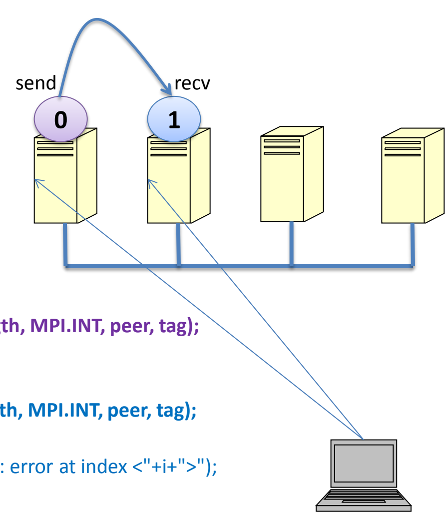
Message Tag

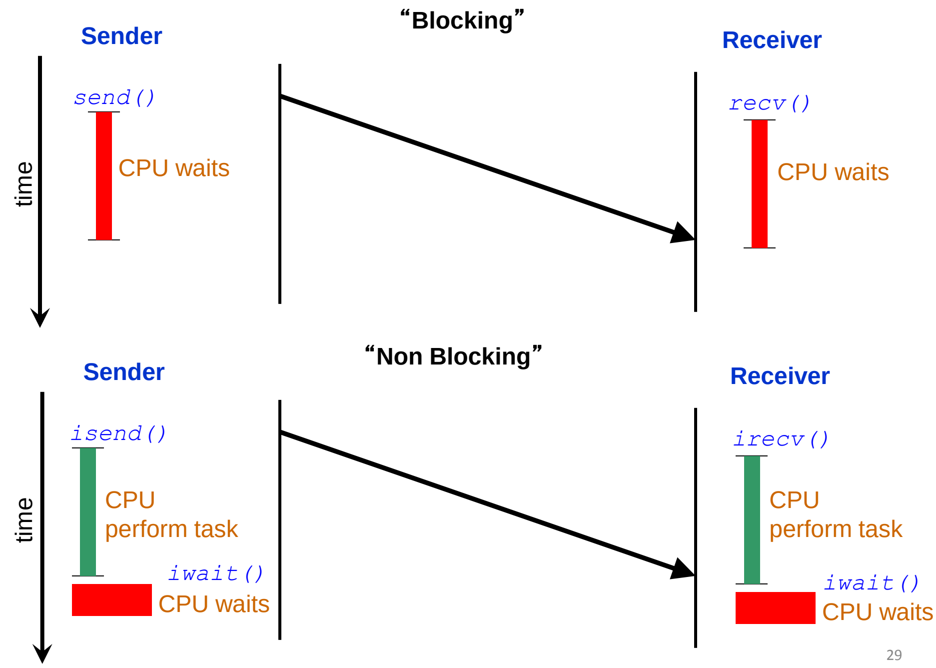
- Used to differentiate between different types of messages being sent

- If special type matching is not required, a wild card message tag MPI.ANY_SOURCE is used, so that the recv() will match with any send().




```
import mpi.*;

public class SimpleSendRecv {
   public static void main(String[] args) {
      int[] buf = new int[10];
      int tag = 10; int peer;

      MPI.Init(args) ;
      int rank = MPI.COMM_WORLD.Rank();

      if (rank == 0) {
            for(int i=0 ; i<buf.length ;i++) buf[i] = i;
            peer = 1;
            MPI.COMM_WORLD.Send(buf, 0, buf.length, MPI.INT, peer, tag);
      } else if (rank == 1) {
            peer = 0;
            MPI.COMM_WORLD.Recv(buf, 0, buf.length, MPI.INT, peer, tag);
            for(int i=0 ; i<buf.length ;i++)
               if(buf[i] != i)   System.out.println(buf[i]+": error at index <"+i+">");
      }
      MPI.Finalize();
   }
}
```





29

Blocking vs Non blocking

Blocking

- A blocking send routine will only "return" after it is safe to modify the application buffer (your send data) for reuse.
- A blocking send can be synchronous
  - handshaking occurring with the receive task confirms a send
- A blocking send can be asynchronous
  - system buffer is used to hold the data
- A blocking receive only "returns" if data received is ready for application.

Non-Blocking

- Non-blocking send and receive routines behave similarly –
  - they will return almost immediately.
  - do not wait for any communication events to complete
- Non-blocking operations simply "request" the MPI library to perform the operation when it is able.
- It is unsafe to modify the application buffer (your variable space)
  - Use "wait" routines to verify if send or received is completed.
- Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains.

30

Non-Blocking Send Example

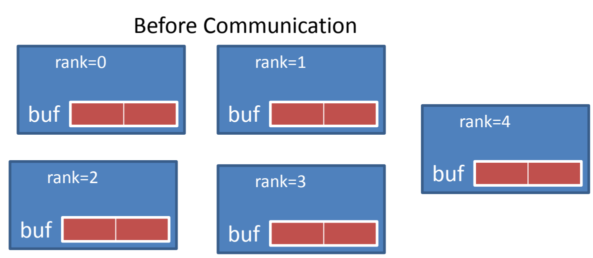
```
Request req =
      MPI.COMM_WORLD.Isend(buffer, offset, count,
                           type, dest, tag) ;
...
req.Wait() ;
```

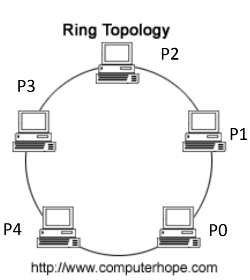
- Immediate communication methods like Isend() return *immediately* with a Request object.
- To wait for completion, execute the Wait() method on that object.
- Effect of Isend/Wait above identical Send, but can do other things (...) in between initiation and waiting.

31

Nearest Neighbor exchange

```
import mpi.*;

public class SimpleSendRecv {
   public static void main(String[] args) {
...
int prev = rank-1;
int next = rank+1;
int buf[2];
int send_buf[1];
if (rank == 0)  prev = numproc - 1;
if (rank == (numproc - 1))  next = 0;
send_buf[0] = rank;

Request r1=MPI.COMM_WORLD.Irecv(buf, 0,  1, MPI_INT, prev, tag1);
Request r2=MPI.COMM_WORLD.Irecv(buf, 1. 1, MPI_INT, next, tag2);

Request s1= MPI.COMM_WORLD.Isend(send_buf, 0, 1, MPI_INT, prev, tag2);
Request s1= MPI.COMM_WORLD.Isend(send_buf, 0, 1, MPI_INT, next, tag1);

// {  do some work  }
r1.wait();
r2.wait();
s1.wait();
s2.wait();

MPI.Finalize();
 }
```

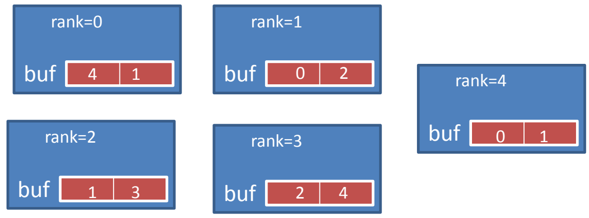




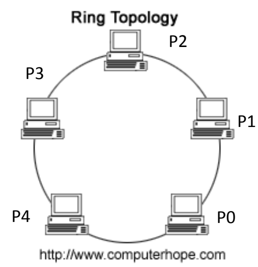
32

Nearest Neighbor exchange

After Communication



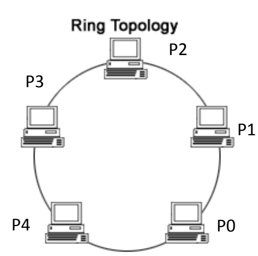

```
import mpi.*;

public class SimpleSendRecv {
   public static void main(String[] args) {
...
int prev = rank-1;
int next = rank+1;
int buf[2];
int send_buf[1];
if (rank == 0)  prev = numproc - 1;
if (rank == (numproc - 1))  next = 0;
send_buf[0] = rank;

Request r1=MPI.COMM_WORLD.Irecv(buf, 0,  1, MPI_INT, prev, tag1);
Request r2=MPI.COMM_WORLD.Irecv(buf, 1. 1, MPI_INT, next, tag2);

Request s1= MPI.COMM_WORLD.Isend(send_buf, 0, 1, MPI_INT, prev, tag2);
Request s1= MPI.COMM_WORLD.Isend(send_buf, 0, 1, MPI_INT, next, tag1);

// {  do some work  }
r1.wait();
r2.wait();
s1.wait();
s2.wait();

MPI.Finalize();
 }
```



33

Nearest Neighbor exchange

```
import mpi.*;

public class SimpleSendRecv {
   public static void main(String[] args) {
...
prev = rank-1;
next = rank+1;
if (rank == 0)  prev = numproc - 1;
if (rank == (numproc - 1))  next = 0;


MPI.COMM_WORLD.Recv(buf[0], 1, MPI_INT, prev, tag1);
MPI.COMM_WORLD.Recv(buf[1], 1, MPI_INT, next, tag2);

MPI.COMM_WORLD.Send(rank, 1, MPI_INT, prev, tag2);
MPI.COMM_WORLD.Send(rank, 1, MPI_INT, next, tag1);


MPI.Finalize();
 }
```



34