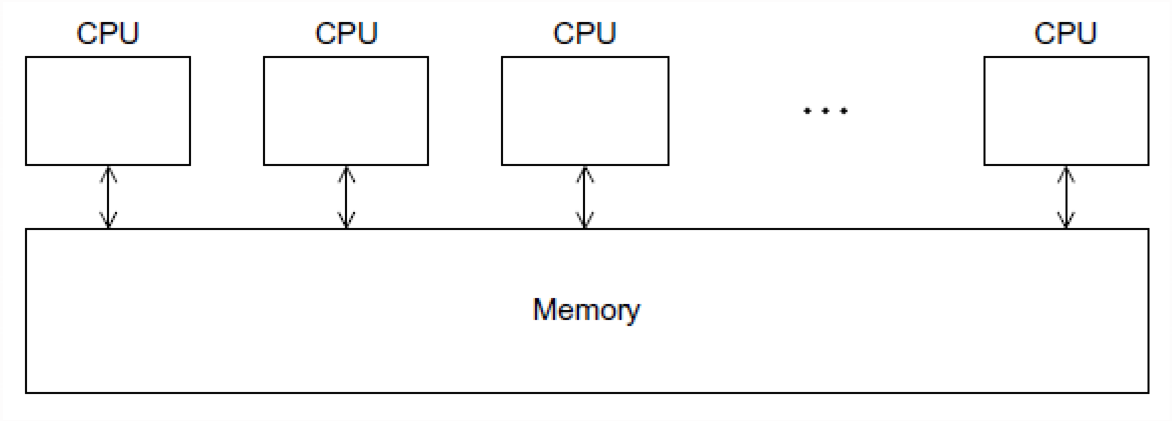
UNIVERSITY OF WOLVERHAMPTON

5CS022 Distributed and Cloud Systems Programming

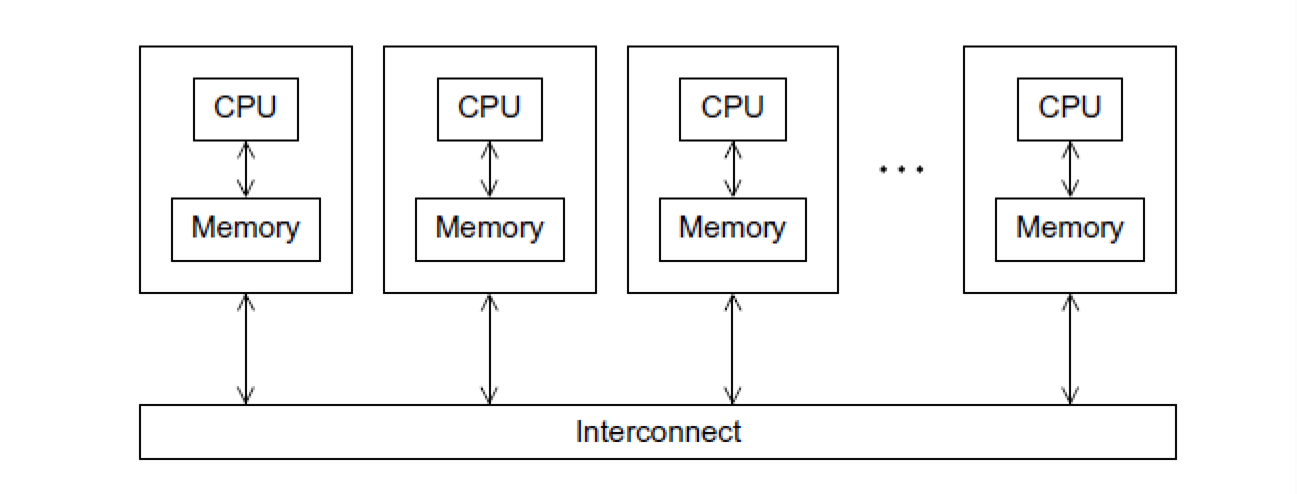
MPI - Message Passing Interface Part 2

Recap: MPI vs Multithreading

- Multithreading (shared memory)



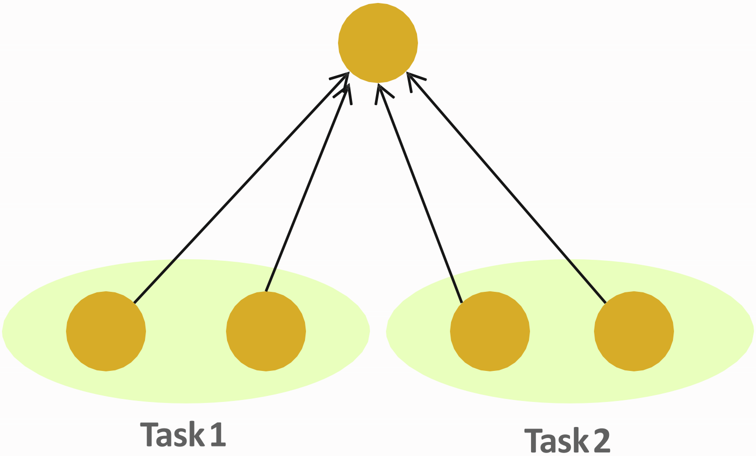
- MPI



Core MPI Functions

- Most MPI programs can be written using just these six core functions:
  - **MPI_Init**
    - `int MPI_Init(int *argc, char ***argv)`
    - initialize the MPI library (must be the first routine called)
  - **MPI_Comm_size**
    - `int MPI_Comm_size(MPI_Comm comm, int *size)`
    - get the size of a communicator
  - **MPI_Comm_rank**
    - get the rank of the calling process in the communicator
  - **MPI_Send**
    - ```
      int MPI_Send(const void *buf, int count, MPI_Datatype datatype,
                   int dest, int tag, MPI_Comm comm)
      ```
    - send a message to another process
  - **MPI_Recv**
    - ```
      int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
                   int tag, MPI_Comm comm, MPI_Status *status)
      ```
    - receive a message from another process
  - **MPI_Finalize**
    - `int MPI_Finalize()`
      clean up all MPI state (must be the last MPI function called by a process)

Typical MPI Example



- Each “worker process” computes some task and sends it to the “supervisor” process together with its group number: the “tag” field can be used to represent the task
  - Data count is not fixed
  - Order in which workers send output to master is not fixed
  - Different workers = different source ranks, and different tasks = different tags

Example code

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

const int MAX_STRING = 100;

int main(void) {
  char        greeting[MAX_STRING];
  int         size;
  int         rank;

  MPI_Init(NULL, NULL);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank != 0) { //Worker node
    sprintf(greeting, "Hello from process %d of %d!", rank, size);
    MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
  }
  else {  // Supervisor node
    printf("Process 0 started\n");
    for (int q = 1; q < size; q++) {
      MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      printf("%s\n", greeting);
    }
  }
  MPI_Finalize();
  return 0;
}
```

STATUS OBJECT

- The status object is used after completion of a receive to find the actual length, source, and tag of a message
- Status object is MPI-defined type and provides information about:
  - The source process for the message (status.MPI_SOURCE)
  - The message tag (status.MPI_TAG)
  - Error status (status.MPI_ERROR)
- The number of elements received is given by:

```
MPI_Get_count(MPI_Status *status, MPI_Datatype
  datatype,int *count)
```

  - status - return status of receive operation
  - datatype - datatype of each receive buffer element (handle)
  - count - number of received elements (integer)(OUT)

Using the "status" field

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

int main(int argc, char **argv)
{
  int         size;
  int         rank;
  int         tag;
  int count;
  MPI_Status status;
  int data[100];

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0){
    for(int i = 0; i < size - 1; i++) {
      MPI_Recv(data, 100, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      MPI_Get_count(&status, MPI_INT, &count);
      printf("Node ID: %d; tag: %d; MPI_Get_count: %d; \n",status.MPI_SOURCE, status.MPI_TAG, count);
    }
  }
  else {
    tag = rank *100;
    MPI_Send(data, rand()%100, MPI_INT, 0, tag, MPI_COMM_WORLD);
  }
  MPI_Finalize();
  return 0;
}
```

Blocking vs. Non-blocking Communication

- **MPI_Send/MPI_Recv** are blocking communication calls
  - Return of the routine implies completion
  - When these calls return the memory locations used in the message transfer can be safely accessed for reuse
  - For "send" completion implies variable or buffer sent can be reused/modified
  - Modifications will not affect data intended for the receiver
  - For "receive" variable received can be read

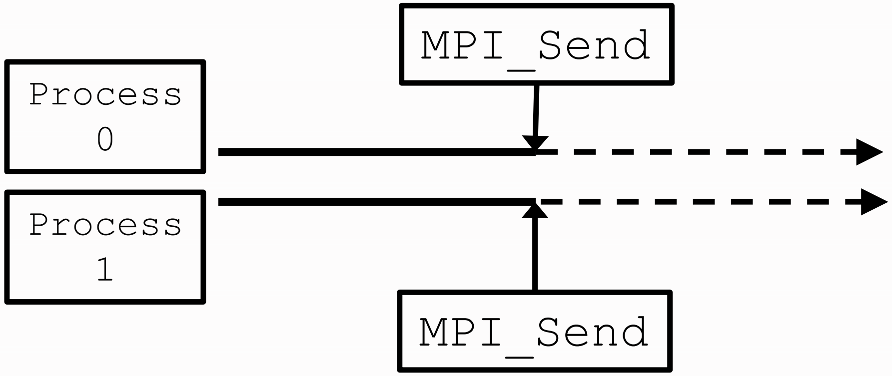
- **MPI_Isend/MPI_Irecv** are non-blocking variants
  - Routine returns immediately – completion has to be separately tested for
  - These are primarily used to overlap computation and communication to improve performance

Blocking Communication

- In blocking communication.
  - MPI_SEND does not return until buffer is empty (available for reuse)
  - MPI_RECV does not return until buffer is full (available for use)

- A process sending data will be blocked until data in the send buffer is emptied
- A process receiving data will be blocked until the receive buffer is filled
- Exact completion semantics of communication generally depends on the
- message size and the system buffer size
- Blocking communication is simple to use but can be prone to deadlocks

```
if (rank == 0) {
  MPI_Send(..)
  MPI_Recv(..)
}
else {
  MPI_Send(..)
  MPI_Recv(..)
}
```



Non-Blocking Communication

- Non-blocking operations return immediately with ‘‘request handles” that can be waited on and queried
  - **MPI_Isend**(start, count, datatype, dest, tag, comm, **request**)
  - **MPI_Irecv**(start, count, datatype, src, tag, comm, **request**)
  - **MPI_Wait**(**request**, status)
- Non-blocking operations allow overlapping computation and communication
- One can also test without waiting using **MPI_Test**
  - **MPI_Test**(request, flag, status)
- Anywhere you use MPI_Send or MPI_Recv, you can use the pair of
  - **MPI_Isend/MPI_Wait or MPI_Irecv/MPI_Wait**
- Combinations of blocking and non-blocking sends/receives can be used to synchronize execution

Multiple Completions

- It is sometimes desirable to wait on multiple requests:

```
MPI_Waitall(count, array_requests, array_statuses)
MPI_Waitany(count, array_requests, &index,&status)
MPI_Waitsome(count, array_requests, array_indices,array_statuses)
```

- There are corresponding versions of test for each of these

Message Completion and Buffering

- For a communication to succeed:
  - Sender must specify a valid destination rank
  - Receiver must specify a valid source rank (including MPI_ANY_SOURCE)
  - The communicator must be the same
  - Tags must match
  - Receiver's buffer must be large enough
- A send has completed when the user supplied buffer can be reused

```
*buf =3;
MPI_Send(buf, 1, MPI_INT …)
*buf = 4; /* OK, receiver will always
receive 3 */
```

```
*buf =3;
MPI_Isend(buf, 1, MPI_INT …)
*buf = 4; /*Not certain if receiver
gets 3 or 4 or anything else */
MPI_Wait(…);
```

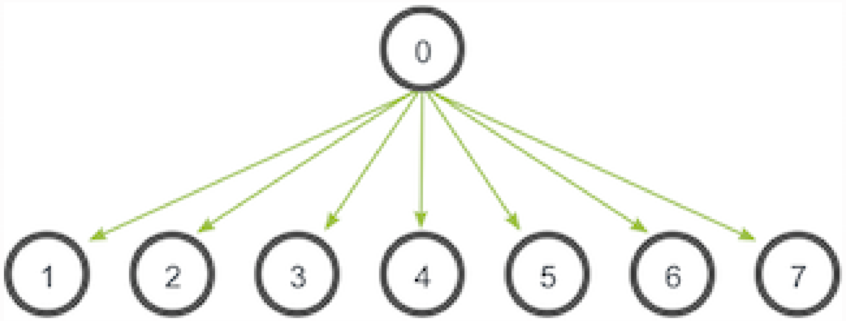
- Just because the send completes does not mean that the receive has completed
  - Message may be buffered by the system
  - Message may still be in transit

A Non-Blocking example

```
int main(int argc, char ** argv)
{
  if (rank == 0) {
    for (i=0; i< 100; i++) {
      /* Compute each data element and send it out */
      data[i] = compute(i);
      MPI_ISend(&data[i], 1, MPI_INT, 1, 0, MPI_COMM_WORLD,
        &request[i]);
    }
    MPI_Waitall(100, request, MPI_STATUSES_IGNORE);
  }
  else {
    for (i = 0; i < 100; i++)
      MPI_Recv(&data[i], 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
  }
}
```

Broadcasting with MPI

- A broadcast is one of the standard collective communication techniques.
- During a broadcast, one process sends the same data to all processes in a communicator.



- One of the main uses of broadcasting is to send out user input to a parallel program, or send out configuration parameters to all processes.

MPI_Broadcast

- The function for broadcast in MPI is MPI_Broadcast()

```
MPI_Bcast(void* data, int count, MPI_Datatype datatype,
          int root, MPI_Comm communicator)
```

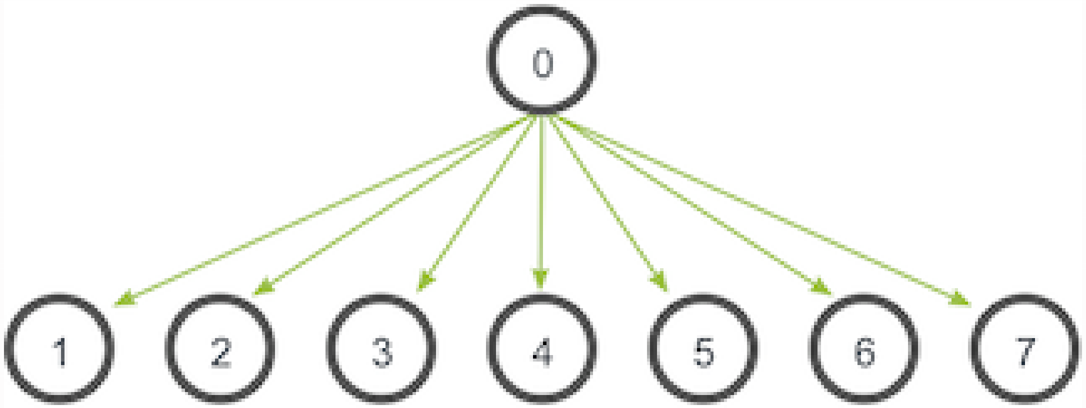
- Although the root process and receiver processes do different jobs, they all call the same MPI_Bcast function.
- When the root process calls MPI_Bcast, the data variable will be sent to all other processes.
- When all of the receiver processes call MPI_Bcast, the data variable will be filled in with the data from the root process.

MPI_Broadcast Example

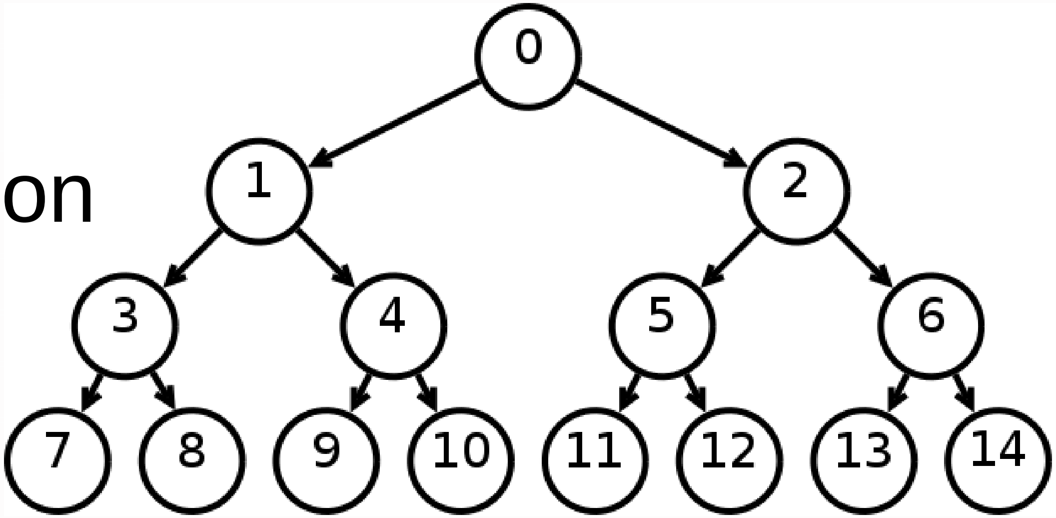
```
int main(int argc, char** argv) {
  int rank;

  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  int message = 0;

  if(rank==0){
    printf("process %d is about to start broadcasting\n", rank);
    message = 42;
  } else {
    printf("process %d is about to receive broadcast\n", rank);
  }
  MPI_Bcast(&message, 1, MPI_INT, 0, MPI_COMM_WORLD);
  if(rank==0){
    printf("process %d has finished broadcasting\n", rank);
  } else {
     printf("process %d has received %d\n", rank, message);
  }
  MPI_Finalize();
}
```

MPI_Bcast Implementation

Logically



Typical implementation



Copying and Running MPI Programs on Other Nodes

- In order to run MPI programs on other nodes, they have to be copied to all the node.
- The usual way is to use "scp":

```
scp mpi01 1098765@remote-node:~/mpi01
```

- Then run it:

```
mpiexec -H localhost,remote-node ./mpi01
```

- However, this will either prompt you for a every time, or it will just not work.
- To make it work, you have to copy your SSH keys to all the nodes

ssh-copy-id

- ssh-copy-id is used to copy and install a publish sign-in key on MPI remote nodes so that the login in automatic.
- First use **`ssh-keygen`** to generate a key in ".ssh/mykey"
- Then copy the key to the remote node:
  `ssh-copy-id -i ~/.ssh/mykey user@remote-node`
- ssh to the remote-node once.
- After that, ssh logins to the remote node do not require logins and MPI will work

End