National University of Computer & Emerging Sciences

Lecture 08

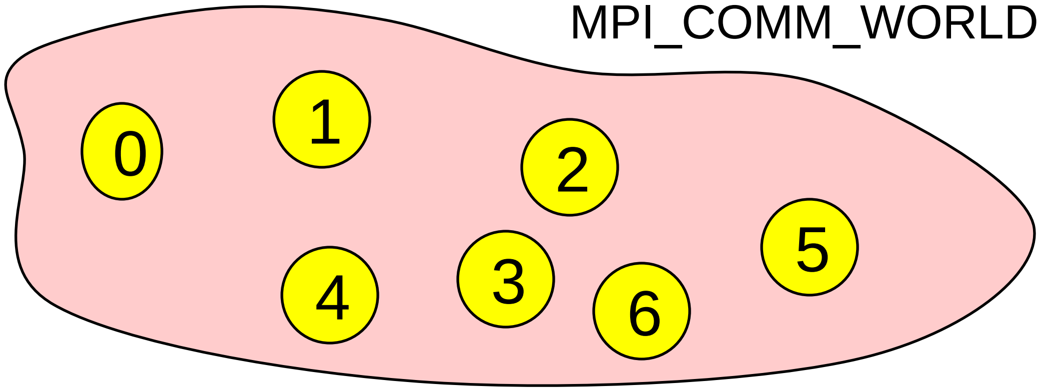
MPI - Point-to-Point Communication

MPI Fundamentals

- A **communicator** defines a group of processes that have the ability to communicate with one another.
- In this group of processes, each is assigned a unique ***rank***, and they explicitly communicate with one another by their ranks.

Communicator MPI_COMM_WORLD

- All processes of an MPI program are members of the default **communicator MPI_COMM_WORLD**.
- MPI_COMM_WORLD is a predefined
- Each process has its own **rank** in a communicator:
  - starting with 0
  - ending with (size-1)



```c
#include <mpi.h>
    {
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD,

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

                                    ROCESSOR_NAME];

          " out of %d processors\n",
          processor_name, world_rank, world_size);

    // Finalize the MPI environment.
    MPI_Finalize();
}
```

MPI header file

MPI_Comm_size returns the size of a communicator. Here, MPI_COMM_WORLD encloses all of the processes, so this call should return the amount of processes that were requested for the job.

MPI environment initialization, all of MPI's global and internal variables are constructed. For example, a communicator is formed around all of the processes that were spawned, and unique ranks are assigned to each process.

MPI_Comm_rank returns the rank of a process in a communicator. Ranks are incremental starting from zero and are primarily used for identification purposes during send/receive.

MPI_Get_processor_name obtains the actual name of the processor on which the process is executing.

clean up the MPI environment

Running MPICH Program

- Regular applications

```
./mpi_hello_world
```

- MPI applications (running with 16 processes)

```
mpiexec –n 16 ./mpi_hello_world
```

Messages

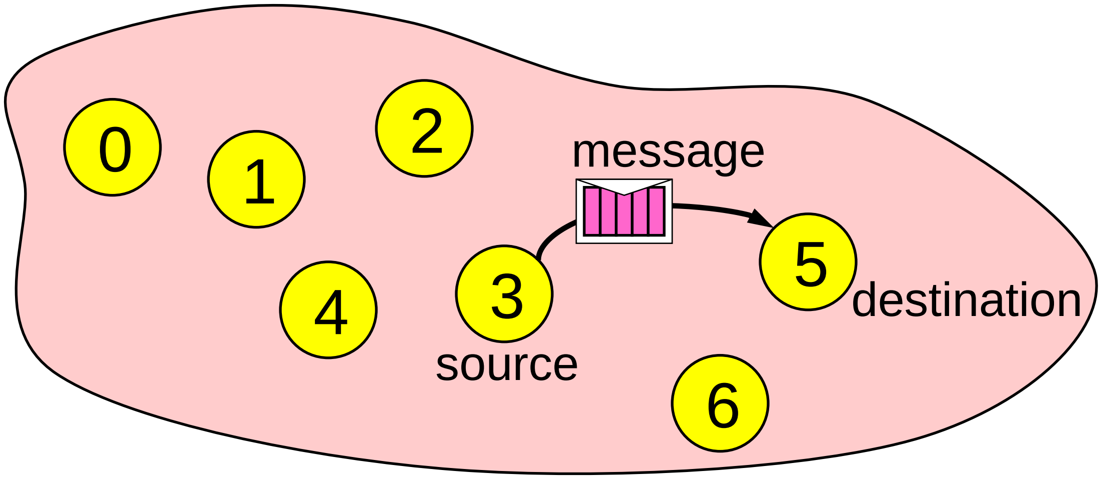
- A message contains a number of elements of some particular datatype.

Example: message with 5 integers

2345	654	96574	-12	7676

Point-to-Point Communication

- Communication between two processes.
- Source process sends message to destination process.
- Communication takes place within a communicator, e.g., MPI_COMM_WORLD.
- Processes are identified by their ranks in the communicator.



Point to Point Communication

- Communication is done using send and receive operations among processes.
  - To send a message, sender provides the rank of the process and a unique *tag* to identify the message.
  - The receiver can then receive a message with a given tag (or it may not even care about the tag), and then handle the data accordingly.
  - Two basic (and simple) functions, MPI_Send and MPI_Recv

Data Communication in MPI

- Communication requires the following information:
  - Sender has to know:
    - Whom to send the data to (receiver's process rank)
    - What kind of data to send (100 integers or 200 characters, etc)
    - A user-defined "tag" for the message (think of it as an email subject; allows the receiver to distinguish different messages)
  - Receiver "might" have to know:
    - Who is sending the data (OK if the receiver does not know; in this case sender rank will be `MPI_ANY_SOURCE`, meaning anyone can send)
    - What kind of data is being received (partial information is OK: I might receive *up to* 1000 integers)
    - What the user-defined "tag" of the message is (OK if the receiver does not know; in this case tag will be `MPI_ANY_TAG`)

MPI Send

```
MPI_Send( void* data, int count, MPI_Datatype datatype, int dest,
          int tag, MPI_Comm comm)
```

- The target process is specified by `dest` and `comm`.
- The data, its type and amount is described by (`data, count, datatype`).
- `tag` is a user-defined “identifier” for the message

- When this function returns, the data has been delivered to the system and the buffer can be reused.
  - The message may not have been received by the target process.

MPI Receive

```
MPI_Recv( void* data, int count, MPI_Datatype datatype, int
    source, int tag, MPI_Comm comm, MPI_Status* status)
```

- The sending process is specified by **`source`** and **`comm`**.
- The receiving data buffer and its type is defined by **`data`** and **`datatype`**.
- Receiving fewer than **`count`** occurrences of **`datatype`** is OK, but receiving more is an error.

- Waits until a matching (on **`source`**, **`tag`, `comm`**) message is received from the system, and the buffer can be used.

Elementary MPI datatypes

- MPI Datatype is very similar to a C datatype
  - **`int`** → **`MPI_INT`**
  - **`double`** → **`MPI_DOUBLE`**
  - **`char`** → **`MPI_CHAR`**
- More complex datatypes are also possible:
  - E.g., you can create a structure datatype that comprises of other datatypes → a char, an int and a double.

- The “count” in **`MPI_SEND`** and **`MPI_RECV`** refers to how many datatype elements should be communicated

Simple Communication in MPI

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

int main(int argc, char ** argv)
{
    int rank;
    char data[] = "Hello World";

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0)
        MPI_Send(data, 11, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
    else if (rank == 1)
        MPI_Recv(data, 11, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
    printf("I am a slave and I received %s message
    from my master", data);

    MPI_Finalize();
    return 0;
}
```

So far so good but …

- Let's review the **MPI_RECV** function

```
MPI_RECV(buf, count, datatype, source, tag, comm, status)
```

- The receiver may not know the rank of source, the tag the receiver used and the *count* of elements sent by receiver.

- What can the receiver do?

Wildcards

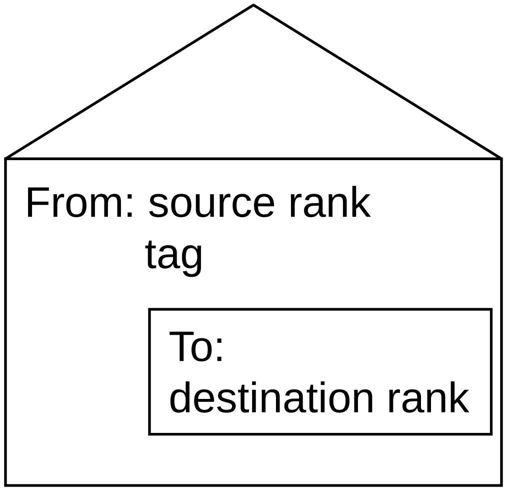
- Receiver can wildcard.
- To receive from any source — source = MPI_ANY_SOURCE
- To receive from any tag — tag = MPI_ANY_TAG
- The count variable is the upper-bound on the number of elements received, it doesn't specify *how many were actually received*.

```
MPI_RECV(buf, SOME_MAX_NUMBER, datatype,
MPI_ANY_SOURCE, MPI_ANY_TAG, comm, status)
```

The MPI_Status structure

- Envelope information is returned from MPI_RECV in *status*.

status.MPI_SOURCE
status.MPI_TAG
count via MPI_Get_count()



The MPI_Status structure

- **`status`** contains further information:
  - Who sent the message (can be used if you used **`MPI_ANY_SOURCE`**)
  - How much data was actually received
  - What tag was used with the message (can be used if you used **`MPI_ANY_TAG`**)
  - **`MPI_STATUS_IGNORE`** can be used if we don't need any additional information

The MPI_Status structure

- The MPI_Recv operation takes the address of an MPI_Status structure as an argument (which can be ignored with MPI_STATUS_IGNORE).
- If we pass an MPI_Status structure to the MPI_Recv function, it will be populated with additional information about the receive operation after it completes.

The MPI_Status structure

- The three primary pieces of information include:
  - **The rank of the sender**. The rank of the sender is stored in the MPI_SOURCE element of the structure. That is, if we declare an MPI_Status stat variable, the rank can be accessed with stat.MPI_SOURCE.
  - **The tag of the message**. The tag of the message can be accessed by the MPI_TAG element of the structure (similar to MPI_SOURCE).
  - **The length of the message**. The length of the message does not have a predefined element in the status structure. Instead, we have to find out the length of the message with MPI_Get_count.

The MPI_Status structure

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

- In MPI_Get_count, the user passes the MPI_Status structure, the datatype of the message, and count is returned. The count variable is the total number of datatype elements that were received.

```c
const int MAX_NUMBERS = 100;
int numbers[MAX_NUMBERS];
int number_amount;
if (world_rank == 0) {
    // Pick a random amont of integers to send to process one
    srand(time(NULL));
    number_amount = (rand() / (float)RAND_MAX) * MAX_NUMBERS;

    // Send the amount of integers to process one
    MPI_Send(numbers, number_amount, MPI_INT, 1, 0, MPI_COMM_WORLD);
    printf("0 sent %d numbers to 1\n", number_amount);
} else if (world_rank == 1) {
    MPI_Status status;
    // Receive at most MAX_NUMBERS from process zero
    MPI_Recv(numbers, MAX_NUMBERS, MPI_INT, 0, 0, MPI_COMM_WORLD,
             &status);

    // After receiving the message, check the status to determine
    // how many numbers were actually received
    MPI_Get_count(&status, MPI_INT, &number_amount);

    // Print off the amount of numbers, and also print additional
    // information in the status object
    printf("1 received %d numbers from 0. Message source = %d, "
           "tag = %d\n",
           number_amount, status.MPI_SOURCE, status.MPI_TAG);
}
```