

Blocking Point-to-Point Communications

Types of Communication

- In MPI there are two main types of communication

Point-to-point communication:

- A communication involving a pair of processes
- Good if a process only needs to communicate with a limited number of other processes

Collective communication:

- A communication involving all the processes
- May seem like point-to-point communication is all that is required as could use a set of them to communicate with everyone
 - Very inefficient – Sending data from one process to every other process involves $p-1$ communications from the originating node
 - Collective communications can do things like relaying data – Original process sends to a subset of the nodes, which, in turn send the data on to other nodes that have not yet received it
 - Communication time scales as approximately $\log(p)$

Blocking point-to-point communication

- We will initially do blocking point-to-point communication
 - Easier to code and shows the basic idea
 - Later on we will look at non-blocking communications

Two main functions involved:

- `MPI_Send` – Sends data to another process
- `MPI_Recv` – Receives data from another process
- Because these are blocking, you must be careful of the send and receive order
 - If a process tries to send data without all the previous receives being handled or vice versa there will be a block in communication and the code will hang or crash
 - We will later look at using probe as a way around this problem so that messages can be received from different processes in any order (amongst other uses)

Simple point-to-point communication

- This program sends data from process zero to all other processes (a different random number sent to each process):

```
#include <mpi.h>
#include <iostream>
#include <cstdlib>
#include <time.h>

using namespace std;

int id, p;

int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    srand(time(NULL)+id*10);

    int tag_num = 1;

    if (id == 0)
    {
        for (int i = 1; i < p; i++)
        {
            int send_data = rand();
            MPI_Send(&send_data, 1, MPI_INT, i, tag_num, MPI_COMM_WORLD);
            cout << send_data << " sent to processor " << i << endl;
            cout.flush();
        }
    }
    else
    {
        int recv_data;
        MPI_Recv(&recv_data, 1, MPI_INT, 0, tag_num, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
        cout << recv_data << " received on processor " << id << endl;
        cout.flush();
    }

    MPI_Finalize();
}
```

Random Numbers

- In a lot of the examples I use random numbers to give me something to send
 - `rand` generates an integer random number between 0 and `RAND_MAX`
 - `srand` seeds the random number generator
 - Both of these functions are found in `<cstdlib>`
- To seed the random number I use `srand(time(NULL)+id*10);`
 - `time(NULL)` gives a time as the number of seconds since the beginning of 1970
 - As all the processes will start at the same time, I try and have them generate different random number by adding their id times 10 – These are still only pseudo-random and so are likely to be related to one another

MPI_Send

```
int MPI_Send(  
    void* data,  
    int count,  
    MPI_Datatype datatype,  
    int destination,  
    int tag,  
    MPI_Comm communicator)
```

- **data** is a pointer to the data to be sent
- **count** is the number of items to be sent
 - Note that this is the number of variables of type **datatype** to be sent, not the number of bytes
- **datatype** is the MPI data type to be sent
- **destination** is the id of process that is to receive the data
- **tag** is an identifier for the communication
- For both **MPI_Send** and **MPI_Recv** the return value indicates error or success

MPI_Recv

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

- **data** is a pointer to the data to be received
- **count** is the number of items to be received
- **datatype** is the MPI data type to be received
- **source** is the id of process from which the data is to be received
- **tag** is an identifier for the communication
- **status** is a pointer to a structure that contains information about the communication
 - Stores information such as the processes involved and the size of the data being sent, as well as any communication error information
 - To ignore use **MPI_STATUS_IGNORE**

Do Worksheet 1 Exercise 1

MPI Types and their C/C++ equivalents

MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	char

- These are the simple [MPI_Datatypes](#) available
 - Later we will look at creating our own MPI data types

More on MPI data types

- Note that MPI_BYTE should be used to send generic data of a known size in bytes
 - E.g. if `Object` is a complex object of fixed size (watch out for objects, for instance, containing pointers!):
`MPI_Send(&Object, sizeof(Object), MPI_BYTE, 0, 0, MPI_COMM_WORLD);`
- MPI_BYTE should also be used to send variables of type `bool`
 - `bool` may only represent true or false (0 or 1), but it is stored in a variable of one byte in size as this is the smallest addressable memory unit

More on tags

- Data can potentially be received out of the expected order
 - Especially true in non-blocking communications
 - Need a method to ensure that the correct communication is being dealt with
 - A “race condition” refers to the situation where communications are undesirably handled in the wrong order
- Each communication is given a tag
- The tag should identify the specific type and/or order of communication
 - Either increment the tag for each communication (what I usually do)
 - ...or have a different tag for each type of communication (though this might run the risk of being out by a whole cycle of communications)
 - ...or do a combination by, for instance, having a unique identifier for each communication with a value between 0 and 100 and then add 100 times the number of communication cycles to this value to produce the tag
- Note that I don’t change the tags in most of the examples because I am showing a single communication
- Tags can be any non-negative value below `MPI_TAG_UB`

MPI return codes

- Most MPI communications give a return value that indicates either success or what went wrong. These are some of those codes:

MPI_SUCCESS

- Communication completed successfully

MPI_ERR_COMM

- Invalid communicator

MPI_ERR_COUNT

- Invalid count argument (usually a negative count – a zero is usually valid)

MPI_ERR_TYPE

- Invalid datatype argument (Either not a valid MPI type or, for types you created yourself, a type that has not been committed – more on this in a later lecture)

MPI_ERR_TAG

- Invalid tag argument

MPI_ERR_RANK

- Invalid source or destination rank. Ranks must be between zero and the size of the communicator minus one

Using a Probe

- A probe allows information about a communication to be read before the communication is completed
 - i.e. before `MPI_Recv` is called in the context of blocking communications
- Two very useful pieces of information can be obtained by doing this
 - Which process has sent the data
 - ...and how much data is being sent
 - You need to be able to assign enough memory to the `data` pointer that is receiving the data
 - You could assign more than the maximum that that will ever be sent, but this is inefficient and you need to know what is the maximum that could be send
 - If you wish to assign the correct amount of memory for the amount of data being sent you can either send a communication saying how much data to expect or you can use a probe to find out

MPI_Probe

- `MPI_Probe` is the function that probes the communication:

```
int MPI_Probe(  
    int source,  
    int tag,  
    MPI_Comm comm,  
    MPI_Status *status)
```

- `source` is the source process
- `tag` for the communication
- `comm` is the communicator
- `status` is a pointer to the structure containing the status information
- Having to specify the `source` and the `tag` does seem to remove some of the utility of the function, but `MPI_ANY_SOURCE` and/or `MPI_ANY_TAG` can be set

Using the information from MPI_Status

```
typedef struct {  
    int count;  
    int cancelled;  
    int MPI_SOURCE;  
    int MPI_TAG;  
    int MPI_ERROR;  
} MPI_Status;
```

- **count** is the number of received entries
 - Use **MPI_Get_count** in conjunction with the status rather than using this variable directly – avoids potential issues with mismatches in variable type or size
- **cancelled** is true or false depending on whether the corresponding communication was cancelled
- **MPI_SOURCE** is the source process for the communication
- **MPI_TAG** is the tag associated with the communication
- **MPI_ERROR** is an error code (there are a lot of potential errors and you can look them up if you want)
 - Has a value of **MPI_SUCCESS** if there is no error

MPI_Get_count

- `MPI_Get_count` is used in conjunction with the status to get the number of items (not bytes) sent:

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

- `status` is a pointer to the status structure
- `datatype` of the data being sent
- `count` is a pointer to an integer that will store the number of items being sent
- Note that the return value is again an error code and not the number of items (which is stored in the variable pointed to by `count`)

Using MPI_Probe to get the size of the data

- Similar to the previous example, with processor zero sending data to each of the other processes – Difference is that it randomly sends between 1 and 5 items

```
#include <mpi.h>
#include <iostream>
#include <cstdlib>
#include <time.h>
```

```
using namespace std;
```

```
int id, p;
```

```
int main(int argc, char *argv[])
{
```

```
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    srand(time(NULL)+id*10);
```

```
    int tag_num = 1;
```

```
    if (id == 0)
    {
```

```
        for (int i = 1; i < p; i++)
        {
```

```
            int num_send = 1 + rand() % 5;
            int *send_data = new int[num_send];
            for (int j = 0; j < num_send; j++) send_data[j] = rand();
```

```
            MPI_Send(send_data, num_send, MPI_INT, i, tag_num, MPI_COMM_WORLD);
```

```
        for (int j = 0; j < num_send; j++) cout << send_data[j] << "\t";
        cout << " sent to processor " << i << endl;
        cout.flush();
```

```
        delete[] send_data;
```

```
    }
```

```
}
```

```
else
```

```
{
```

```
    int *recv_data;
```

```
    int num_recv;
```

```
    MPI_Status status;
```

```
    MPI_Probe(0, tag_num, MPI_COMM_WORLD, &status);
```

```
    MPI_Get_count(&status, MPI_INT, &num_recv);
```

```
    recv_data = new int[num_recv];
```

```
    MPI_Recv(recv_data, num_recv, MPI_INT, 0, tag_num, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
```

```
    for (int j = 0; j < num_recv; j++) cout << recv_data[j] << "\t";
```

```
    cout << " received on processor " << id << endl;
```

```
    cout.flush();
```

```
}
```

```
MPI_Finalize();
```

```
}
```

Do Worksheet 1 Exercise 2

Using MPI_Probe to get the source of the data

- `MPI_Probe` can be used to both wait for data to be sent and to find out where it has come from
 - Note that `MPI_Probe` can time out (you can check for this as it will give an error) and, if you wish to, probe again to continue waiting
 - Note that `MPI_Recv` could directly wait for a message from any process, but it is often useful to prepare to receive data from a specific process
- The following example plays multi-processor ping-pong, with a processor receiving a communication and then passing it on to another random process
- I exit when one process has received 10 messages
 - As other processes don't know how many messages have been received by another process I exit messily using `MPI_Abort` (likely to result in error messages from other processes).
- Processor zero starts the ping-pong
- You will notice that some of the outputs will be out of the logical order
 - The process sends it `cout` back to originating process and so may arrive back in a different order to which the processes generated them – Watch out for this problem when trying to debug parallel code

```
#include <mpi.h>
#include <iostream>
#include <cstdlib>
#include <time.h>

using namespace std;

int id, p;
int tag_num = 1;

void Send_Random_Data(void)
{
    int to_proc;
    while ((to_proc = rand() % p) == id); //Stop code sending to itself
    int send_data = id;

    MPI_Send(&send_data, 1, MPI_INT, to_proc, tag_num, MPI_COMM_WORLD);

    cout << "Processor " << id << " sent data to processor " << to_proc << endl;
    cout.flush();
}
```

MPI_Probe example continued

```
void Recv_Data(void)
{
    int from_proc;
    int recv_data;
    MPI_Status status;

    while (MPI_Probe(MPI_ANY_SOURCE, tag_num, MPI_COMM_WORLD, &status)
           != MPI_SUCCESS)
        cout << "Communication timed out or failed! " << id << endl;

    from_proc = status.MPI_SOURCE;

    MPI_Recv(&recv_data, 1, MPI_INT, from_proc, tag_num, MPI_COMM_WORLD,
            &status);

    cout << "Processor " << id << " received data for processor " << from_proc << endl;
    cout.flush();
}
```

```
int main(int argc, char *argv[])
{
    int recv_cnt = 0;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    srand(time(NULL)+id*10);

    if (id == 0) Send_Random_Data();    //Starts the ping-pong

    while (true)
    {
        Recv_Data();                    //Waits for and receives data
        recv_cnt++;
        if (recv_cnt<10)
            Send_Random_Data();         //Sends it on to a new processor
        else break;                     //Exits if 10 communications are received
    }

    cout << endl << "Process " << id << " has received 10 communications and is
        aborting (the exit will be messy!)" << endl;
    cout.flush();

    MPI_Abort(MPI_COMM_WORLD, 0);
}
```

What is wrong with blocking communications?

- The main problem with blocking communications is that it is hard to do them efficiently if a lot of communications is required
 - Either you can let each process communicate in turn
 - Lots of idle time waiting for other processes to finish their communications
 - ...or you can try and do clever ordering of the communications so that all processes get to do communications at the same time
 - Easy to get wrong – Will block and crash if, for instance, two process are trying to send data to one another or receive data from one another at the same time
 - Still likely to be inefficient as even efficient ordering will rely on, for instance, an assumption that all communications take about the same time
- Things can be much more efficient if communications don't block one another
 - MPI provides the tools to do this
 - More complex to implement, but potentially much more efficient code

Do Worksheet 1 Exercise 3