Non-blocking Point to Point Communications

MPI_Isend and MPI_Irecv

```
int MPI_Isend(
    const void *buf,
    int count,
    MPI_Datatype datatype,
    int dest,
    int tag,
    MPI_Comm comm,
    MPI_Request *request)
```

```
int MPI_Irecv(
    void *buf,
    int count,
    MPI_Datatype datatype,
    int source,
    int tag,
    MPI_Comm comm,
    MPI_Request * request)
```

- You will notice that MPI_Isend and MPI_Irecv are very similar to MPI_Send and MPI_Recv. There are two difference:
  - They both include a request variable – more on that later
  - MPI_Irecv does not have a status variable – this is because the receive is not blocking and so MPI_Irecv actually exits before the communication is necessarily finished and so we do not know the status of the communication at this point

An example with non-blocking sends and receives

```
#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)
	{
		MPI_Request* request=new MPI_Request[p-1];
		int *send_data = new int[p-1];

		for (int i = 1; i < p; i++)
		{
			send_data[i - 1] = rand();
			MPI_Isend(&send_data[i-1], 1, MPI_INT, i, tag_num, MPI_COMM_WORLD, &request[i-1]);
			cout << send_data[i-1] << " sent to processor " << i << endl;
			cout.flush();
		}
		MPI_Waitall(p - 1, request, MPI_STATUS_IGNORE);

		delete[] send_data;
		delete[] request;
	}
	else
	{
		int recv_data;
		MPI_Request request;
		MPI_Irecv(&recv_data, 1, MPI_INT, 0, tag_num, MPI_COMM_WORLD, &request);

		MPI_Wait(&request, MPI_STATUS_IGNORE);

		cout << recv_data << " received on processor " << id << endl;
		cout.flush();
	}
	MPI_Finalize();
}
```

Non-blocking communication

- At first glance this might seem to be doing exactly the same thing as the first example in the blocking communication lecture
  - Overall it does achieve the same thing, but some important differences:
    - The MPI_Isend and MPI_Irecv exit straight away without waiting for the communications to complete
    - The data will be sent in the background. You can carry out computations while the data is being sent:

```
MPI_Irecv(&recv_data, 1, MPI_INT, 0, tag_num, MPI_COMM_WORLD, &request);
//You can do stuff here that doesn't require the communication to have finished
MPI_Wait(&request, MPI_STATUS_IGNORE);
```

    - The data will be sent from processor zero to whichever process is first able to receive the data
    - Note that I have an array to store the send data. This is so that all values are still available until MPI_Waitall

MPI_Wait and MPI_Waitall

- While it is useful to do things while waiting for the communications to finish, at some point you need to be able ensure that a communication has completed so that you can use the data that has been sent
    - This is what MPI_Wait and MPI_Waitall achieves
    - MPI_Wait is used to wait for a single communication to complete
    - MPI_Waitall is used to wait for a list of communications to complete
- Note that MPI_Wait and MPI_Waitall are blocking for the specific communications involved
    - It will continue once its specific communications are finished, irrespective of communications involving other processes

MPI_Wait and MPI_Waitall

```
int MPI_Wait(
    MPI_Request *request,
    MPI_Status *status)

int MPI_Waitall(int count,
    MPI_Request *request_list,
    MPI_Status *status_list)
```

- request is a pointer to a single request object
  - This must match a single MPI_Isend or MPI_Irecv
- status is a pointer to an object which will receive the communication status
  - This can be MPI_STATUS_IGNORE if you do not need the status information

- count is the number of communications that MPI_Waitall will be processing
  - These can be a combination of MPI_Isend and MPI_Irecvs
- request_list is an array that contains count request objects, one for each communication
- status_list is an array of status objects in which the status of each communication will be stored
  - This can again be MPI_STATUS_IGNORE

Sending from everyone to everyone

- This is now easy to achieve using non-blocking communications, but this is still less efficient than a collective operation (which we will cover later)
  - Typically do non-blocking communications such as this where processes are communicating with a subset of the other processes and where these subsets overlap
    - Domain decomposition is an example of this type of problem

Sending from everyone to everyone

```
#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;

	MPI_Request* request = new MPI_Request[(p - 1)*2];
	double *send_data = new double[p];
	double *recv_data = new double[p];

	int cnt = 0;

	for (int i=0;i<p;i++)
		if (i != id)
		{
			MPI_Irecv(&recv_data[i], 1, MPI_DOUBLE, i, tag_num, MPI_COMM_WORLD, &request[cnt]);
			cnt++;
		}
		else recv_data[i] = 0;

	for (int i = 0; i<p; i++)
		if (i != id)
		{
			send_data[i] = (double)id / (double)p;
			MPI_Isend(&send_data[i], 1, MPI_DOUBLE, i, tag_num, MPI_COMM_WORLD, &request[cnt]);
			cnt++;
		}
		else send_data[i] = 0;

	MPI_Waitall(cnt, request, MPI_STATUS_IGNORE);

	for (int i = 0; i < p; i++)	cout << "Processor " << id << " recieved " << recv_data[i] << " from processor " << i << endl;

	delete[] request;
	delete[] recv_data;
	delete[] send_data;

	MPI_Finalize();
}
```

A couple of things to notice…

- With the MPI_Isend I had to store the values of each of the data points as a separate variable in an array
    - This is because the data can be sent anytime until the MPI_Waitall is called
    - This means that the data musn't be change/overwritten or go out of scope within this interval
    - This is different to MPI_Send, where the data to be sent is buffered and/or actually sent by the time MPI_Send completes
    - The same is true of MPI_Irecv, though this is more obvious as you want to have the data
- You will notice that I set up the receives before the sends. This is because data can be sent as soon as there is a matching pair of sends and receives
    - Therefore slightly more efficient to have receive ready and waiting for a send
    - Possibly even more efficient to have receives and sends interleaved with an ordering such that some of the communications can get started while others are still being set up
        - Alternatively you can set up the receives, do some calculations, and then set up the sends
    - Remember that all the non-blocking communications are occurring in the background on a separate communications thread

Do Worksheet 2 Exercise 1

Checking if communications are open

- In non-blocking communications it sometimes useful to check if the communications are finished without requiring them to be finished
  - For instance, you could have a loop in which you do some work that does not require the data communication to be finished, with the loop continuing until the communication is finished
  - If you are repeatedly sending more data to a process to do calculations on, you could create the next receive as soon as the previous send has been do and then check if the communication has completed, more data being sent as soon as it completes
    - This should be the basis for worksheet 2 workshop exercise 1
- There are two functions for achieving this – MPI_Test and MPI_Testall

MPI_Test and MPI_Testall

```
int MPI_Test(
    MPI_Request *request,
    int *flag,
    MPI_Status *status)

int MPI_Testall(int count,
    MPI_Request *request_list,
    int *flag,
    MPI_Status *status_list)
```

- These functions are similar to MPI_Wait and MPI_Waitall in terms of the required parameters
- The difference is the flag parameter, which is a pointer to an integer that will be 1 or 0 (true or false) depending on whether the communications associated with request or request_list have completed or not
  - Note that in MPI_Testall the flag will true only if all the communications are finished

Example using MPI_Test and MPI_Testall

- In this example we send a lot of data and do things (albeit not very useful things!) while waiting for the communication to finish

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

using namespace std;

int id, p;

void Do_Work(void)        //Some (not very useful) work
{
        int sum = 0;
        for (int i = 0; i < 100; i++) sum = sum + 10;
}

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, sent_num = 100000, cnt = 0, flag = 0;

 if (id == 0)
{
        MPI_Request* request = new MPI_Request[p - 1];
        int **send_data = new int*[p - 1];
        for (int i = 0; i < p - 1; i++)
        {
                send_data[i] = new int[sent_num];
                for (int j = 0; j < sent_num; j++)
                        send_data[i][j] = rand();
        }

        for (int i = 1; i < p; i++)
                MPI_Isend(send_data[i-1], sent_num, MPI_INT, i, tag_num, MPI_COMM_WORLD, &request[i - 1]);

        while (MPI_Testall(p - 1, request, &flag, MPI_STATUS_IGNORE) == MPI_SUCCESS && flag == 0)
        {
                Do_Work();
                cnt++;
        }
```

Example using MPI_Test and MPI_Testall

```
            for (int i = 0; i < p - 1; i++)
                delete[] send_data[i] ;
            delete[] send_data;
            delete[] request;
        }
        else
        {
            int *recv_data=new int[sent_num];
            MPI_Request request;
            MPI_Irecv(recv_data, sent_num, MPI_INT, 0, tag_num, MPI_COMM_WORLD, &request);

            while (MPI_Test(&request, &flag, MPI_STATUS_IGNORE) == MPI_SUCCESS && flag == 0)
            {
                Do_Work();
                cnt++;
            }

            delete[] recv_data;
        }

        cout << "Process " << id << " did " << cnt << " cycles of work while wating " << endl;
        cout.flush();

        MPI_Finalize();
}
```

- Note that I don't actually need to use the MPI_Wait or MPI_Waitall as I know that all the communications are complete when the while loop exits
- The reason why I have (MPI_Test(&request, &flag, MPI_STATUS_IGNORE) == MPI_SUCCESS && flag == 0) as the condition is so that I can both call MPI_Test and check the value of the flag within the condition
    - In C/C++ conditions are evaluated from left to right and therefore the MPI_Test is done first
    - The check for MPI_SUCCESS is not in expectation of it not being a success, but rather so that the function can be called – If the return is not a success as it implies a communication failure and the program is likely to crash anyway

Do Worksheet 2 Exercise 2

Communications between all processes

- We have shown that non-blocking communications help when sending data between multiple processes
  - Also allows you to do things while waiting
  - Will also continue as soon as a communication is finished – communications don't need to "wait their turn"
- Non-blocking communications are generally better than blocking communications
  - Best alternative when communicating with a number of "neighbouring processes", but not all the processes
    - E.g. processes connected as graph
    - This is what will be the case in, for instance, domain decomposition – more on this later
- If you want to communicate between all processes at the same time collective communications are generally best

Do Worksheet 2 Exercise 3