Parallel Programming: Worksheet 1

Exercise 1: Ring Communication

Write a program that does a ring communication. Each process should send a new number to the next process in addition to the numbers received from the previous process. In other words, process zero should send a single number to process 1, process 1 should send two numbers to process 2 (the original one from the first process together with a new number), process 2 should then send 3 numbers to process 3 etc.. This should continue until the last process sends the entire list of numbers back to the first process. At each stage the process should give its id and the number that it is adding to the list, with process zero then outputting the entire list when it gets back to it. Note that each process should do a single send and a single receive, with an array being sent between the processes.

Exercise 2: Ring Communication with MPI_Probe

Modify the program from exercise 1 so that each process randomly adds between 1 and 3 new numbers to the list being passed around the ring. You should probe for the size of the data to be received, allocate enough memory to store both this data and the new data to be added to the list, receive this data, append the new data to the end of the dynamically allocated array and then send this data on to the next processor. Remember to free the memory after using it.

Exercise 3: Everyone speaking to everyone else

Write a program where every processor sends a piece of information to every other processor using MPI_Send and MPI_Recv.

a) The easiest, but slowest way to do this is to allow each process to have a turn to send data, with each other process waiting for that data (if you have 100 processes they will by waiting 99% of the time and only communicating 1% of the time on average). Implement this method.
b) It is much quicker to allow every process to be sending or receiving data at the same time. You can either achieve this with clever ordering of the sends and receives or by using probe to handle communications as they come in (though you still need to watch out for potential blocking – We will be looking at non-blocking communications later. These are a bit more complex to implement, but remove the need to order communications to avoid blocking). See if you can implement this using blocking communications. Don’t worry if it crashes when you first try, it is easy to get something wrong here!

This exercise is not about how you would implement this in practice (you would always use collective communications if all processes are involved and non-blocking sends and receives if a subset of processes are involved), but rather to get you to think about how blocking can occur in these types of communications.

Note that with some MPI implementations the code can sometimes not block when you expect it to block. This is because implementations often use non-blocking communications “under-the-hood” of their blocking communication implementations – don’t rely on this, though, as you might suddenly find your code not working if you move to another compiler/operating system or when you try and send much larger amounts of data.

Workshop Exercise 1

Write a MPI program in which node zero creates 10,000 pairs of random integers. These should then be divided amongst the processes, with each process being sent their portion of the lists using MPI_Send/MPI_Recv pair. The processes should then calculate the Greatest Common Divisor (GCD, largest number that that each of the numbers is exactly divisible by) for each of the pairs using Euclids Algorith. Once the calculations are complete the results should all be sent back to processor zero, which they should be stored in a single array.

You can check if it all worked by taking the modulus of each of the pairs of numbers with the returned solution and ensure that the result is always zero (this is actually checking that the result is a divisor, though not necessarily the greatest one).