Parallel Programming: Worksheet 5

Exercise 1: Creating memory windows and MPI_Put and MPI_Get operations for a Gather

Write the gather equivalent of the scatter operation that we did in class using one sided communications. You can decide whether you want to use MPI_Put or MPI_Get to achieve this (or try both). Overall each process should create a chunk of data and at the end this data should be stored in processor order on process zero. In this example you should synchronise the communications using MPI_Win_fence.

Remember that the memory that needs to be made remotely accessible will depend upon whether you are using the put or the get operation.

Exercise 2: A Gather operation using PSWC

Rewrite exercise 1, but this time synchronise the communications using PSWC.

Do the exercise twice, based once on MPI_Put and then based on MPI_Get.

Exercise 3: Passive synchronisation using MPI_Win_lock and MPI_Win_unlock

Rewrite the class example where a MPI_Win_lock and MPI_Win_unlock is used to maintain a counter. Instead of having each process generate a new random number after obtaining a counter, generate all the random numbers upfront on process zero in remotely accessible memory. Allocate enough memory to store a counter as well as all the random numbers. You can make the counter the first item in the memory and the random data in the subsequent memory.

At each cycle the process needs to read off the random number at the location in the memory on process zero corresponding to the counter. You will need to use MPI_Win_flush to ensure that the counter read by the MPI_Get is valid before using it to retrieve the random number. This can be done within the same lock/unlock epoch.

See if you can modify the code so that the process pulls over and processes a bigger chunk of data at each cycle. Could you also get the code to write the answers back to process zero (i.e. similar to the GCD example from previous lectures)?

Exercise 4: Implement a reduce operation using MPI_Accumulate

Gather and reduce operations are similar to one another. Convert the gather program that you wrote in exercise 2 to a reduce program. This does mean, though, that instead of the receive buffer on the root process being the size of the send buffer times the number of processes, it need only be the same size as the send buffer, with each process accumulating to the same memory locations.

Workshop Exercise 5

Do the same communication pattern as found in Worksheet 2 question 3, but using one-sided communications. In other words you should divide your *p* processes into a grid of size *m* x *n* (try and ensure that *m* and *n* are integers that are as close to one another as possible. E.g. if *p* is 9, *m* and *n* should both be 3, while if *p* is 12, one should be 3 and the other 4). On this grid calculate an *i* and *j* index for each process such that $id = i + m\,j$.

Each process should communicate with the processes that next to it vertically, horizontally and diagonally (i.e. processes in the middle of the grid will communicate with 8 neighbours, those on edges with 5 neighbours and those in the corners with 3 neighbours). Each process should store the source's *id* as well as their *i* and *j* coordinates in the shared memory of their neighbour (you will need to design a pattern for storing this memory and it might be easiest to have shared memory available for communications from neighbours in each direction even if those neighbours don't exist (pre-store -1s to identify those direction from which no information was received.

Do the synchronisations using PSWC. As each process will be both sending information and receiving information, the best will be to do both MPI_Win_post and MPI_Win_start on every process (of course then followed be an MPI_Win_wait and an MPI_Win_complete).

Once all the communications are complete each process should display the information that is in their shared memory (what they have obtained from the other processes). Not that I have written this description assuming the each process will doing an MPI_Put to send data to their neighbours. The communications could also be achieved using MPI_Get where each process reads the information off their neighbours.

Note that this is again exactly the communication pattern if you were to implement the coursework problem using RMA and one-sided communications.