Parallel Programming: Worksheet 6

Exercise 1: Data Decomposition: Output

Write a program that does matrix multiplication in parallel. On processor zero create 2 matrices to be multiplied (ensure that they are of compatible dimensions for multiplication). Make each of the processes responsible for different rows in the answer.

a) Send the input matrices to all the processes together with the rows that they are responsible for (decide the decomposition on processor zero). Once the calculations are complete gather the results back on processor zero.
b) You do not actually need to send all of both matrices to other processes. Can you speed up the communications by only sending the required data to each process?

Exercise 2: Domain Decomposition

Solve the problem from the MNM lecture on potential flow in parallel:

An electrical current is flowing into a bar of height 0.5m and length 2m (assume that it is deep enough in the 3rd dimension for this to be considered a 2D problem). On the left hand side there is a 2D electrical flux of 10A/m. This flux splits evenly out of the top and bottom of the bar, with the outflowing flux decreasing linearly down to zero at the right hand end of the top and bottom boundaries. There is no flux out of the right hand side of the bar. You can solve this using stream functions and Dirichlet boundary conditions.

Implement an Jacobi iteration algorithm to solve this problem in parallel using domain decomposition on a 101x401 grid. You can do a striped decomposition where you divide the domain into strips. Remember that you need to swap the data at the edge of the strip on every iteration.

Exercise 3: Quicksort Algorithm

The standard serial version of the quicksort algorithm is as follows:

1) Divide the current list into two portions using a pivot value where every item smaller than the pivot goes into one list and every item greater than or equal to the pivot goes into the second list. Unless you know something special about the distribution of the data you can use the midpoint of the data range as the pivot (on the first division you may need to find or be told the range. On subsequent divisions you will already know the range). Alternatively you can simply use the value at the end of the list as a pivot.
2) Do step one recursively until you only have one item in the list.
3) On the way out of the recursion recombine the lists to result in a list that is sorted.

I want you to implement two parallel versions of the quicksort algorithm. There are two basic tactics that you can use in the parallel decomposition of the quicksort algorithm:

a) Have one version where instead of a single pivot on the initial division you have p-1 pivots, where p is the number of processes. This will result in p lists, with one processor responsible for each of the sub-lists. The potential inefficiency in this method is that processor zero is responsible for all of the pivoting and the division of the data between the processes is unlikely to be very even unless the input data is very uniform.
b) The second tactic is to have 2 pivots on each processor. The first processor sends data to two other processors and keeps some of the data for itself (the amount it keeps for itself should be approximately the total amount of data divided by the number of processors).

Each of these two processors should do the same thing until there are no processors left. The sorted lists then need to be regathered back in the reverse order.
The sorting of each list on an individual processor is best done recursively. For distributing the processes it is best to keep track of the processes that a given process has available to distribute. This can be sent along with the list of numbers that that process is responsible for. You may need to use MPI_ANY_SOURCE to receive the list to be sorted by that process. You can then get the number of the source from the status in order to send the sorted list back to the appropriate location.

Exercise 4: Parallel Efficiency

In this task I wish you to investigate the parallel efficiency of a program. A good one to test would be the second version of the Worksheet 2 Workshop Exercise (the reason I suggest the second version is that it will run on a single core as well) or exercise 1 or 2 from this worksheet. Alternatively, you can test any of the programs that you have written, though ideally choose a problem where you can vary both the size of the problem and the number of processes used (and which will run on a single core so that you can estimate the serial cost without writing a dedicated serial code).

You will need to add some timing code to you program. To be fairer in your analysis your code should ideally not write too many couts within the portion of the code that you are timing (or any at all – In fact cout should generally be kept to a minimum in parallel codes unless debugging due to the communications involved).

As the number of cores on your laptops will be quite limited, test this efficiency using the HPC system. Note how the efficiency changes as you go from using a single node to using more than one node.

Use the timings to calculate the parallel efficiency and the speedup ratio as a function of the number of cores. How does this change with the size of the problem that you are testing?

Note that because you are using random numbers as your input the time taken will change from run to run even for the same size of problem and number of cores. There are two ways around this problem – Either run the code multiple times and use the average time taken or you can give a set number rather than the time as the seed for the random number generator so that it always generates the same set of random numbers.

Workshop Exercise 6: Exploratory decomposition

Implement the solution of the 15 puzzle problem in parallel. With only 4 processes the implementation is relatively straight forward as each of the processes can be responsible for one of the moves from the initial state, with all the subsequent moves carried out on those processes. With more available nodes the decomposition is more complex as each of the process needs to send new moves on to available processes while keeping some of the solution for itself. At each step there needs to be a communication between the nodes to determine if a solution has been found. If one is found the chain of moves required needs to be sent back to the root.