High Performance Computing Workshop 3

January 31, 2021

Exercise 2

In this exercise we create some parallel code to test out Message Passing Interface (MPI) functions using the cirrus network. All the code was written in C++. We write code to calculate $A \times B$ where A and B are the $N \times N$ matrices defined

$$A_i, j = (N - j + i + 1)i$$
$$B_{i,j} = (j + i)(N - j + 1).$$

We will use parallel code using MPI functions to split the task between N processes. The root process will compute the matrix B and then broadcast it to the other processes, each of which will compute a single row of A and multiply this by B to to find a row of $A \times B$ each. The root process will then gather these rows and concatenate the individual rows into a whole matrix $A \times B = C$ which it will print.

The code to compute the matrix C in serial form is relatively straightforward. The key parts we will focus on are the broadcasting of the information from the root process to the other processes and the gathering of information by the root process.

To achieve this in C++ we include the library mpi.h. This allows us to access the MPI library of function calls. We initialise the MPI by calling

$$MPI\_Init(Null, Null)$$

and set the MPI communicators to MPI_COMM_WORLD. This ensures that all available processes will be communicating with each other, since in this task we need each process to be in the same communicator as each other process.

Each process is assigned a rank, a number $0, 1, .., N - 1$ which we will use to assign the correct row calculation to that process. Meaning that process k will be assigned the task of computing row k. The process of rank 0 is defined to be the root processes.

The function $MPI\_Comm\_rank(comm, \&rank)$ gives us the rank of each process that is running the code.

We ensure that the root process alone calculates the matrix B by using an if statement, i.e. $if(rank == 0)$, and stores it in multidimensional array.

When this is completed we call

$$MPI\_Bcast(B, N * N, MPI\_INT, 0, MPI\_COMM\_WORLD)$$

to broadcast the matrix B to all other processes. In this function B is being broadcast, $N * N$ is the size of the information being broadcast, MPI_INT is the type of data, 0 is the rank of the root process and MPI_COMM_WORLD is ensuring the data is sent to every process in the communicator.

Each process then receives the matrix B and calculates an array $Crow$, the row corresponding to the rank of the process.

Each process then calls

$$MPI\_Gather(\&Crow, N, MPI\_INT, C, N, MPI\_INT, 0, MPI\_COMM\_WORLD)$$

so that the root process can gather all of the rows. Crow ensures that this is the information being sent back to the root process, N is the size of the information being sent, MPI_INT is the integer type of information, C is where the root process will store all the gathered information, N is the size of the information received from each other process, MPI_INT is the type of information that the received information will be saved as, 0 is to ensure that the information is gathered by the root process and MPI_COMM_WORLD ensures that all processes are gathered from.

Then using another if statement we ensure that only the root process prints out the the completed matrix.

To ensure that each process is in fact computing the correct corresponding row our code also prints out the row it has computed followed by "I'm from rank k" before it sends the row back to the root processes. We complete the MPI by calling

$$MPI\_Finalize()$$

to finish the script.

To run the script we specify in our slurm file that we will use a single node and N tasks per node, so long as $N < 36$. Otherwise we use some combination such that the number of nodes multiplied by the number of tasks per node is equal to N.

Exercise 3

In this exercise we are tasked with performing a double integration. We use MPI parallelisation to split the domain of integration into many smaller domains, each can be computed by a single process.

We compute this problem with a double integral given by

$$\int_0^{100} \int_0^{100} x \sin(x^2) + y \sin(y^2) dx dy \approx 195.215536$$

The problem will however work with any other integral inserted into the script.

The integral is computed using a trapezium like method, which in serial is very computationally heavy since the function oscillates so much. In fact it was so computationally heavy that I was unable to run the code to find a reasonable grid size that would be the required estimated integral to within 3 decimal places of the true integral.

The MPI parts of the script works in a very similar way to the previous exercise with slight differences. This time we allow each process to compute the integral on a small strip of the domain, with the domain divided equally amongst the total number of processes allocated.. Then use MPI_Reduce to add all of these individual integrals together and return them to the root process. The root process will then print out the final integral.

The reducing of the information about the integral is the key component here. The function used was

$$MPI\_Reduce(\&answer, \&int\_total, N, MPI\_DOUBLE, MPI\_SUM, 0, comm).$$

Here "answer" is the integral computed on each strip by each node, int_total is the double that the sum will be added to in the root process, MPI_DOUBLE is the type of information we wish to sum, MPI_SUM is the operation to be competed, 0 is the root process and comm ensures that the communicator includes all of the processes.

I didn't have time to do much more than this this week, but the MPI code gives promising results and appears to be working to give the correct approximation of the integral. For instance when the grid is $10^5 \times 10^5$ and we start 36 processes the code finishes in about 2 minutes and gives an integral that is within 1 of the true integral.