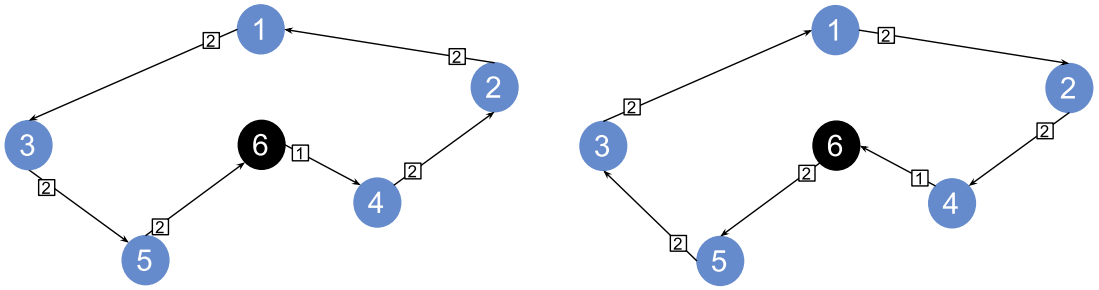
In this assignment, you are asked to make modifications to your first assignment. You are also asked to implement one more travelling salesman algorithm which will be explained. Finally, you are also asked to expand your program using MPI to run multiple instances of each algorithm for all possible starting points. Again, you are encouraged to start this assignment as early as possible to avoid the queues on Barkla. If you want to make use of more Barkla resources, then you will need to start very early.

1 The Travelling Salesman Problem (TSP)

Just like in the previous assignment, we are looking at algorithms which can find an approximation for the Travelling Salesman Problem. Depending which vertex you start at, it can affect the final tour and even return different tour costs.



(a) Cheapest Insertion starting from 1 (b) Cheapest Insertion starting from 6

Figure 1: A comparison of the final tour depending where you started the algorithm.

Figure 1(a)'s tour is (1, 3, 5, 6, 4, 2, 1). Figure 1(b)'s tour is (6, 5, 3, 1, 2, 4, 6). While the resulting cost of the tour is the same and looks like it has been reversed, the tour is different. With a larger set of vertices, this can change the resulting tour cost and the final tour.

1.1 Terminology

We will call each point on the graph the **vertex**. There are 6 vertices in Figure 1.

We will call each connection between vertices the **edge**.

We will call two vertices **connected** if they have an edge between them.

The sequence of vertices that are visited is called the **tour**.

A **partial tour** is a tour that has not yet visited all the vertices.

We will refer to the weights of all the edges added up in the final or partial tour as the **tour cost**.

We will refer to the number of vertices in a tour or partial tour as the **tour length**

2 The solutions

2.1 Preparation of Solution

Just like the previous assignment, you are given a file of coordinates.

x, y
4.81263062736921, 8.34719930253777
2.90156816804616, 0.39593575612759
1.13649642931556, 2.27359458630845
4.49079099682118, 2.97491204443206
9.84251616851393, 9.10783427307047

Figure 2: Format of a coord file

Each line is a coordinate for a vertex, with the x and y coordinate being separated by a comma. You will need to convert this into a distance matrix.

0.000000	8.177698	7.099481	5.381919	5.087073
8.177698	0.000000	2.577029	3.029315	11.138848
7.099481	2.577029	0.000000	3.426826	11.068045
5.381919	3.029315	3.426826	0.000000	8.139637
5.087073	11.138848	11.068045	8.139637	0.000000

Figure 3: A distance matrix for Figure 2

To convert the coordinates to a distance matrix, you will need make use of the euclidean distance formula.

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \tag{1}$$

Figure 4: The euclidean distance formula

Where: d is the distance between 2 vertices v_i and v_j, x_i and y_i are the coordinates of the vertex v_i, and x_j and y_j are the coordinates of the vertex v_j.

2.2 Cheapest Insertion

The cheapest insertion algorithm begins with two connected vertices in a partial tour. Each step, it looks for a vertex that hasn't been visited and inserts it between two connected vertices in the tour, such that the cost of inserting it between the two connected vertices is minimal.

The steps to Cheapest Insertion can be found on the first assignment brief. Assume that the indices i, j, k etc. are vertex labels, unless stated otherwise. In a tiebreak situation, always pick the lowest index (indices).

2.3 Farthest Insertion

The farthest insertion algorithm begins with two connected vertices in a partial tour. Each step, it checks for the farthest vertex not visited from any vertex within the partial tour, and then inserts it between two connected vertices in the partial tour where the cost of inserting it between the two connected vertices is minimal.

The steps to the farthest insertion algorithm can be found on the first assignment brief. Assume that the indices i, j, k etc. are vertex labels unless stated otherwise. In a tiebreak situation, always pick the lowest index (indices).

2.4 Nearest Addition

The nearest addition algorithm begins with two connected vertices in a partial tour. Each step, it looks for the vertex that's nearest to a vertex in the partial tour, and adds it to either side of the of the vertex in the partial tour.

These are the steps to the Nearest Addition algorithm. Assume that the indices i, j, k etc. are vertex labels, unless stated otherwise. In a tiebreak situation, always pick the lowest index (indices).

1. Start off with a vertex v_i.

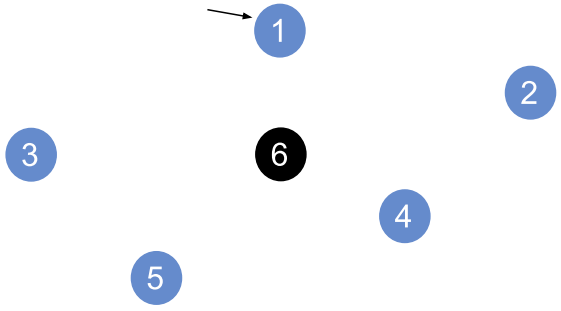


Figure 5: Step 1 of Nearest Addition

2. Find a vertex v_j such that $dist(v_i, v_j)$ is minimal, and create a partial tour (v_i, v_j, v_i)

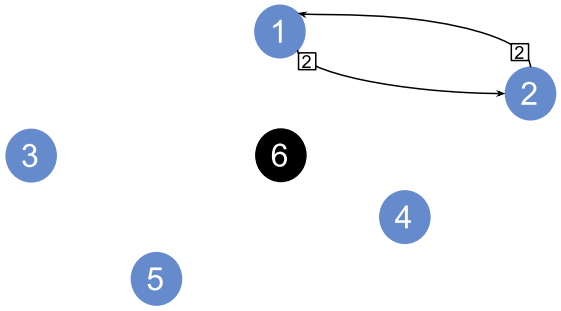


Figure 6: Step 2 of Nearest Addition

3. For all vertices v_n in the partial tour where n is a position in the tour, find a vertex v_k such that $dist(v_n, v_k)$ is minimal.

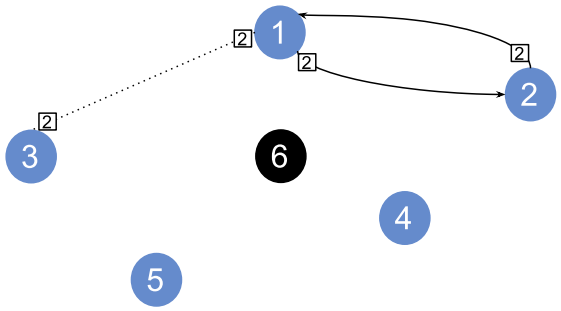


Figure 7: Step 3 of Nearest Addition

4. Add v_k either between v_{n-1} and v_n or between v_n and v_{n+1} depending which one is cheaper. If v_k is closest to the start/end vertex, you must consider adding it to the position after the start position, or the position before the end of the partial tour. In the event of a tie, always add it to the lowest index.

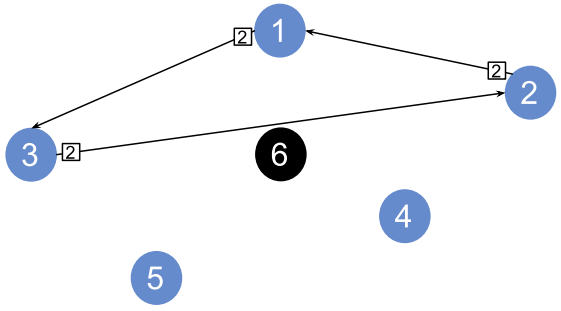


Figure 8: Step 4 of Nearest Addition

5. Repeat steps 3 and 4 until all vertices have been visted, and are in the tour.

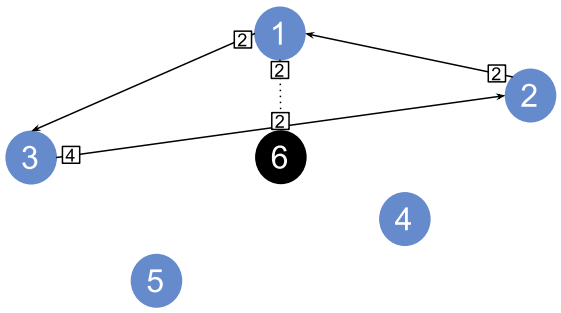


Figure 9: Step 3(2) of Nearest Addition

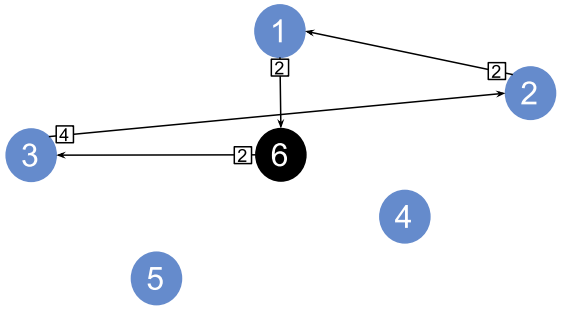


Figure 10: Step 4(2) of Nearest Addition

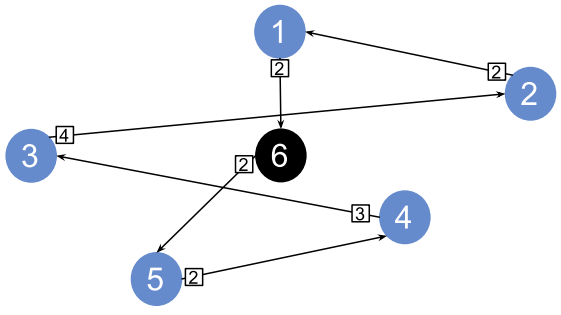


Figure 11: Final Step and tour for Nearest Addition. Tour Cost = 15

3 Distributed Implementation

You are to implement a distributed implementation using MPI that can find the lowest costing tour for each algorithm. Each algorithm should be run for as many vertices as there

are, with each run of an algorithm starting from a different vertex. For example, if there are 4 vertices $v_{0..3}$, all algorithms should run 4 times each. One run starting from v_0, another starting from v_1, another from v_2, and then finally the last run from v_3. The idea is that if you had 4 MPI processes, each process could run 4 instances of Cheapest Insertion (for example) from different starting points. For this example, there would be 12 runs of the algorithms. The Distributed Implementation should run your OpenMP implementations of Cheapest Insertion, Farthest Insertion, and Nearest Addition. In a tiebreak situation, ensure you pick the first lowest costing tour you found.

3.1 Running your programs

Your programs should be able to be run like so:

```
mpirun –np <p> ./<program>.exe <coord_file> <out1> <out2> <out3>
```

Where **p** is the number of processes, **program** is the name of your compiled MPI program, **coord_file** is the name of the coordinate file and **out1** is the output file for your Cheapest Insertion algorithm, **out2** is the output file for your Farthest Insertion tour, and **out3** is the output file for your Nearest Addition tour.

Therefore, your programs should accept a coordinate file, and 3 output files as arguments. Note that C considers the first argument as the program executable.

Your MPI program should read the data file, run the 3 algorithms, testing for every starting point on each, find the lowest costing tour for each algorithm depending where it started, and write the lowest costing tour of each algorithm to the respective output files.

3.2 Provided Code

You are provided with code that can read the coordinate input from a file, and write the final tour to a file. This is located in the file **coordReader.c**. You will need to include this file in your compiler command. Declare the functions at the top of your main file, and don't do #include"coordReader.c" as this has strange behaviour on CodeGrade.

The function **readNumOfCoords()** takes a filename as a parameter and returns the number of coordinates in the given file as an integer.

The function **readCoords()** takes the filename and the number of coordinates as parameters, and returns the coordinates from a file and stores it in a two-dimensional array of doubles, where coords[i][0] is the x coordinate for the ith coordinate, and coords[i][1] is the y coordinate for the ith coordinate.

The function **writeTourToFile()** takes a tour, the tour length, and the output filename as parameters, and writes a tour to the given file.

4 Instructions

- Implement an parallel solution for Nearest Addition. Call this file ompnAddition.c
- Implement a serial solution to the Distributed Implementation, that will execute all OpenMP versions of the algorithms and starting points sequentially. Call this main-openmp-only.c
- Implement a parallel solution to the Distributed Implementation that can execute all OpenMP versions of the algorithms and their starting points in parallel. Call this main-mpi.c
- Create a Makefile and call it "Makefile" which performs as the list states below. Without the Makefile, your code will not grade on CodeGrade (see more in section 5.1).
  - **make gserial** compiles main-serial.c coordReader.c ompcInsertion.c ompfInsertion.c and ompnAddition.c into gserial with the GNU MPI compiler
  - **make gcomplete** compiles main-mpi.c coordReader.c ompcInsertion.c ompfInsertion.c and ompnAddition.c into gcomplete with the GNU MPI compiler
  - **make iserial** compiles main-serial.c coordReader.c ompcInsertion.c ompfInsertion.c and ompnAddition.c into iserial with the Intel MPI compiler
  - **make icomplete** compiles main-mpi.c coordReader.c ompcInsertion.c ompfInsertion.c and ompnAddition.c into icomplete with the Intel MPI compiler
- Test each of your parallel solutions using 1, 2, 4, 8, 16, and 32 OpenMP threads with just one MPI process, recording the time it takes to solve each process. Record the start time after you read from the coordinates file, and the end time before you write to the output file. **Do all testing with the large data file.**
- Using the fastest number of threads, test with 1, 2, 4, 8, 16, and 32 MPI Processes. (Note that to run 32 MPI Processes on just the course node alone, you will need to run with 1 OpenMP thread. To see how you can request more nodes for running, look at the section 5.2)
- Plot a speedup plot for the number of threads using 1 MPI process to solve your whole program.
- Plot a strong scaling plot for the number of MPI processes you use. **Make sure you explicitly state how many OpenMP threads you used**
- Write a report that, for your parallel version of Nearest Addition and your distributed implementation (both serial and parallel) describes your parallelisation strategy.

- In your report, include: the speedup and strong scaling plots, how you conducted each measurement and calculation to plot these, and screenshots of you compiling and running your program. **These do not contribute to the page limit**
- **Your final submission should be uploaded onto CodeGrade. The files you NEED to upload should be:**
  - **Makefile**
  - **ompcInsertion.c**
  - **ompfInsertion.c**
  - **ompnAddition.c**
  - **main-serial.c**
  - **main-mpi.c**
  - **report.pdf**

5 Hints

You can also parallelise the conversion of the coordinates to the distance matrix.

When declaring arrays, it's better to use dynamic memory allocation. You can do this by...

```
int *oned_array = (int *)malloc(numOfElements * sizeof(int));
```

5.1 Makefile

You are instructed to use a MakeFile to compile the code in any way you like. An example of how to use a MakeFile can be used here.

```
{make_command}: {target files}
    {compile_command}

complete: main-mpi.c coordReader.c ompcInsertion.c ompfInsertion.c ompnAddition.c
    mpicc -fopenmp -std=c99 main-mpi.c coordReader.c ompcInsertion.c
    ompfInsertion.c ompnAddition.c -o complete.exe -lm
```

Now, in the Linux environment, in the same directory as your Makefile, if you type 'make ci', the compile command is automatically executed. It is worth noting, the compile command must be indented. The target files are the files that must be present for the make command to execute. Avoid the -Wall flag when compiling, otherwise if there are any warnings, your code will fail to compile and CodeGrade won't run your code.

5.2 Running on more than one node

On Barkla, you can use more than just the course node. The reason we tell you to use the course node is because you can execute without having to wait for other users of Barkla (there are a lot). If you want to execute on more than one node, you can use this sbatch command.

```
sbatch –N <numNodes> –n <numTasks> –c <numCores> –p nodes
```

By changing the value of -N, we can increase the number of nodes we are using, giving you more resources. On the course partition, we are limited to 1 node with 40 cores. If you wanted to run 32 MPI processes with 32 OpenMP threads, you would need 1024 cores meaning you should request 32 nodes. Ensure you use -p nodes otherwise you will request 32 nodes within the course partition, for which there is only one node, and it will be left in a "pending" state. Note that you may have to wait several hours (or days/weeks) for your job to have the resources to run, meaning that if you have left the assignment to the last minute, or the job still hasn't run, test up to 32 MPI processes with just 1 OpenMP thread on the course partition.

5.3 Contiguous Memory

With MPI Collective communications, they can only distribute the memory if it is contiguous (if the memory locations are right next to each other). When trying to pass any array that isn't one-dimensional, there is no guarantee on contiguity, and so MPI will have a problem distributing the array to other processes. To avoid this, you can set up an array as a one-dimensional array, and then reshape it using the following code.

```
double (*2D_Array)[num] = (double(*)[num]) 1D_array;
```

Assuming 1D_array is an array with num * num elements, this piece of code reshapes a one-dimensional array into a two-dimensional array. So whereas 1D_array can be accessed by doing 1D_array[i], you can access 2D_array by 2D_array[i][j]. Note that they share the same memory location, but the compiler will interpret access to memory differently, so any change made to 2D_array will also affect 1D_array and vice versa.

6 Marking scheme

Code that compiles without errors or warnings	15%
Same numerical results for test cases	20%
Speedup plot	10%
Strong scaling plot	10%
Parallel efficiency up to 32 threads	10%
Parallel efficiency up to 32 processes	15%
Clean code and comments	10%
Report	10%

Table 1: Marking scheme

The purpose of this assessment is to develop your skills in analysing numerical programs and developing parallel programs using OpenMP and MPI. This assessment contributes to 35% of this modules grade. The scores from the above marking scheme will be scaled down to reflect that. Your work will be submitted to automatic plagiarism/collusion detection systems, and those exceeding a threshold will be reported to the Academic Integrity Officer for investigation regarding adhesion to the university's policy https://www.liverpool.ac.uk/media/livacuk/tqsd/code-of-practice-on-assessment/appendix_L_cop_assess.pdf.

7 Deadline

The deadline is the 8th January 2024

https://www.liverpool.ac.uk/aqsd/academic-codes-of-practice/code-of-practice-on-assessment/

[END]