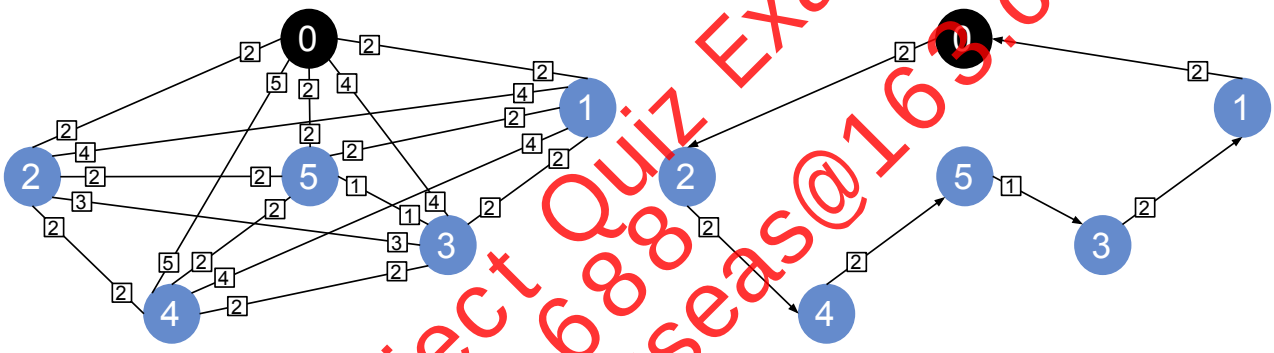
In this assignment, you are asked to implement 2 algorithms for the Travelling Salesman Problem. This document explains the operations in detail, so you do not need previous knowledge. You are encouraged to begin work on this as soon as possible to avoid the queue times on Barkla closer to the deadline. We would be happy to clarify anything you do not understand in this report.

1 The Travelling Salesman Problem (TSP)

The travelling salesman problem is a problem that seeks to answer the following question: 'Given a list of vertices and the distances between each pair of vertices, what is the shortest possible route that visits each vertex exactly once and returns to the origin vertex?'.



(a) A fully connected graph (b) The shortest route around all vertices

Figure 1: An example of the travelling salesman problem

The travelling salesman problem is an NP-hard problem, that meaning an exact solution cannot be solved in polynomial time. However, there are polynomial solutions that can be used which give an approximation of the shortest route between all vertices. In this assignment you are asked to implement 2 of these.

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**. There are 15 edges in Figure 1.
We will call two vertices **connected** if they have an edge between them.
The sequence of vertices that are visited is called the **tour**. The tour for Figure 1(b) is (0, 2, 4, 5, 3, 1, 0). Note the tour always starts and ends at the origin vertex.
A partial tour is a tour that has not yet visited all the vertices.

2 The solutions

2.1 Preparation of Solution

You are given a number of coordinate files with this format:

x, y
4.81263062736921, 8.34719930253777
2.90156816804616, 0.39593575612759
1.13649642931556, 2.27359458630845
4.49079099682118, 2.97491204443206
9.84251616851393, 9.1078342730747

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	2.577629	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.

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

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 Smallest Sum Insertion

The smallest sum insertion algorithm starts the tour with the vertex with the lowest index. In this case that is vertex 0. Each step, it selects a currently unvisited vertex where the total edge cost to all the vertices in the partial tour is minimal. It then inserts it between two connected vertices in the partial tour where the cost of inserting it between those two connected vertices is minimal.

These steps can be followed to implement the smallest sum insertion 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.

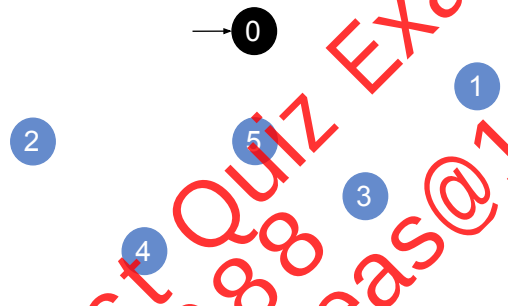


Figure 5: Step 1 of Smallest Sum Insertion

2. Find a vertex v_j such that $\sum_{t=0}^{t=Length(partialtour)} \text{dist}(v_t, v_j)$ is minimal.

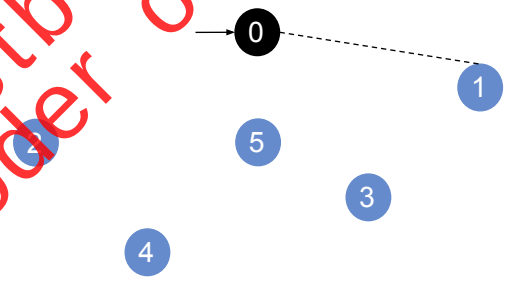


Figure 6: Step 2 of Smallest Sum Insertion

3. Insert v_j between two connected vertices in the partial tour v_n and v_{n+1}, where n is a position in the partial tour, such that $\text{dist}(v_n, v_j) + \text{dist}(v_{n+1}, v_j) - \text{dist}(v_n, v_{n+1})$ is minimal.

4. Repeat steps 2 and 3 until all of the vertices have been visited.

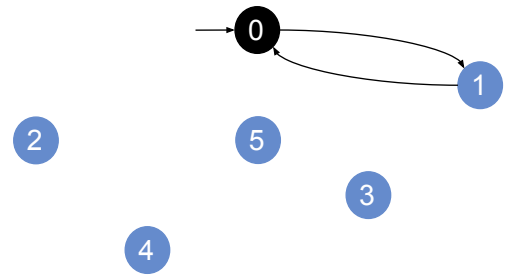
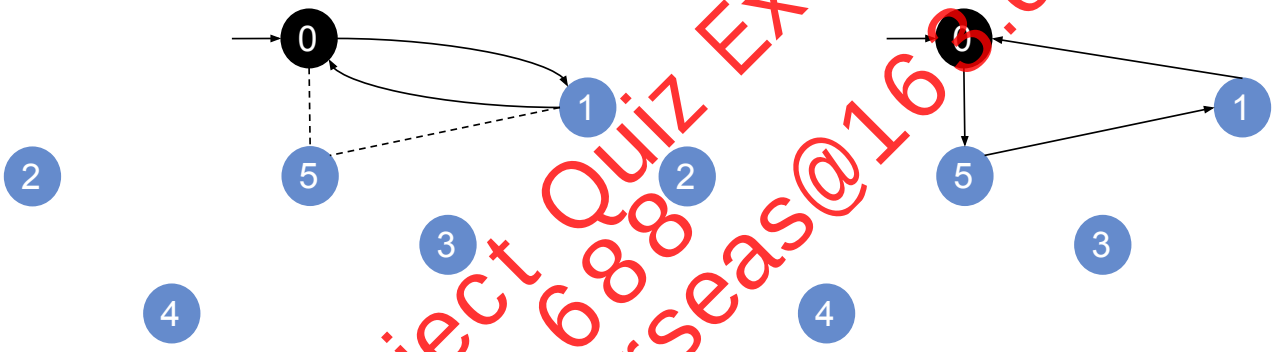
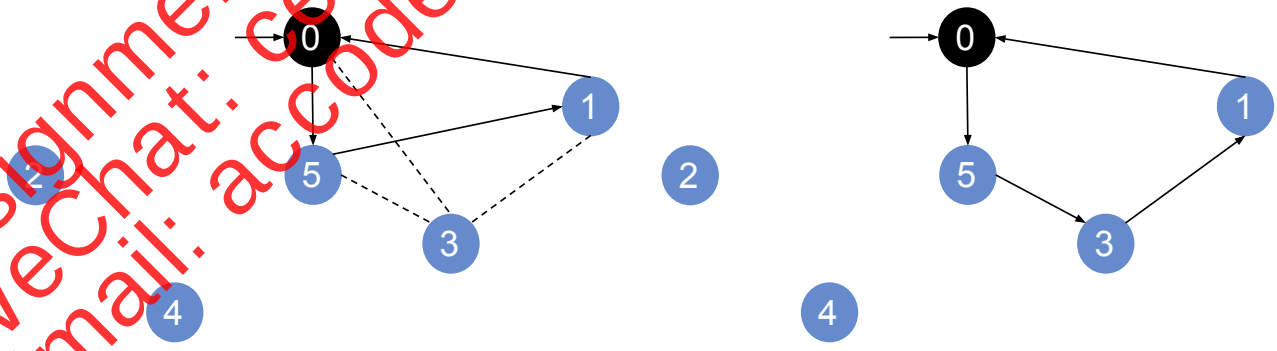


Figure 7: Step 3 of Smallest Sum Insertion



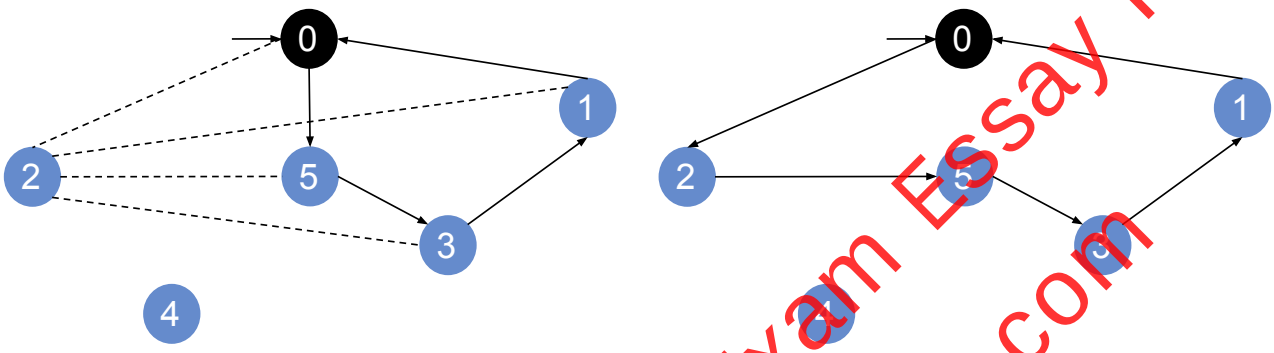
(a) Select the vertex (b) Insert the vertex

Figure 8: Step 4 of Smallest Sum Insertion



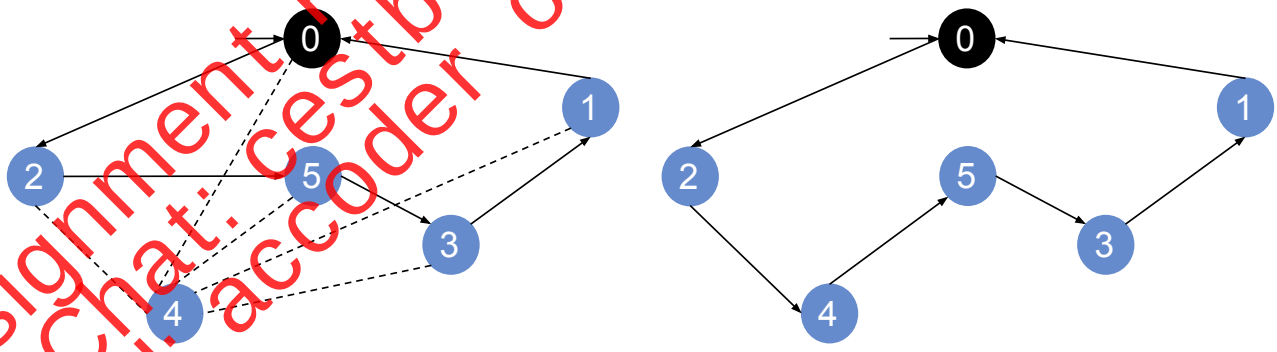
(a) Select the vertex (b) Insert the vertex

Figure 9: Step 5 of Smallest Sum Insertion



(a) Select the vertex (b) Insert the vertex

Figure 10: Step 6 of Smallest Sum Insertion



(a) Select the vertex (b) Insert the vertex

Figure 11: Step 7 of Smallest Sum Insertion

2.3 MinMax Insertion

The minmax insertion algorithm starts the tour with the vertex with the lowest index. In this case that is vertex 0. Each step, it selects a currently unvisited vertex where the largest edge to a vertex in the partial tour is minimal. It then inserts it between two connected vertices in the partial tour where the cost of inserting it between those two connected vertices is minimal.

These steps can be followed to implement the minmax insertion 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.

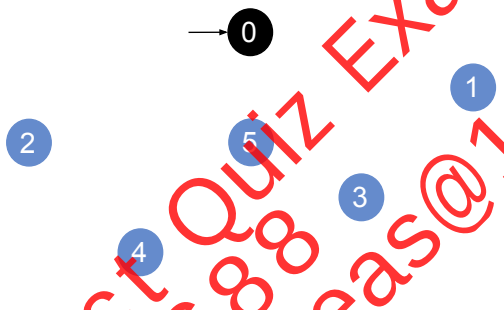


Figure 12: Step 1 of Minmax Insertion

2. Find a vertex v_j such that $Max(dist(v_t, v_j))$ is minimal, where t is the list of elements in the tour.

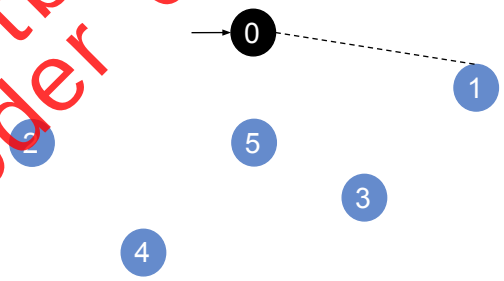


Figure 13: Step 2 of Minmax Insertion

3. Insert v_j between two connected vertices in the partial tour v_n and v_{n+1}, where n is a position in the partial tour, such that dist(v_n, v_j) + dist(v_{n+1}, v_j) - dist(v_n, v_{n+1}) is minimal.

4. Repeat steps 2 and 3 until all of the vertices have been visited.

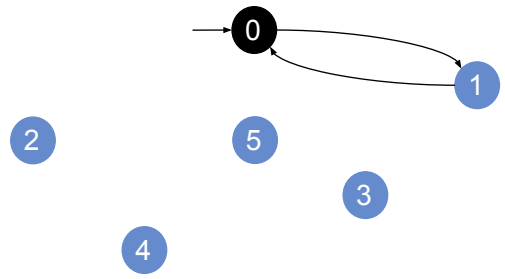
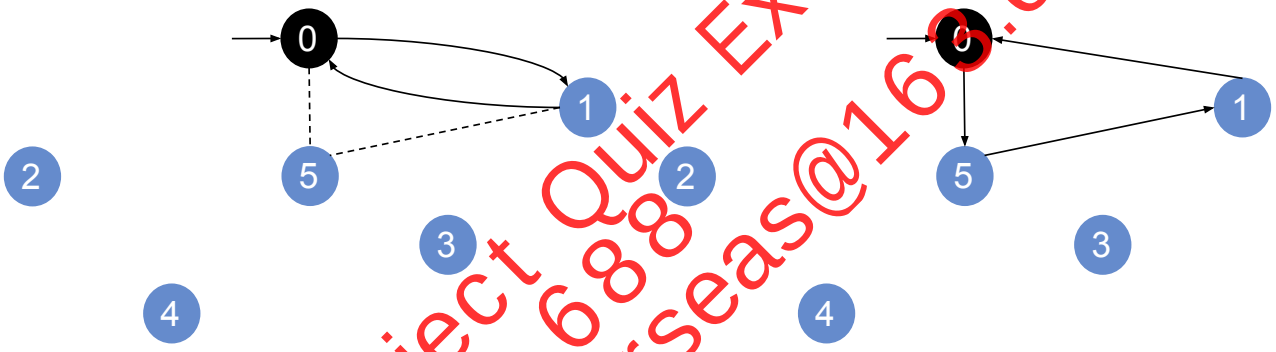
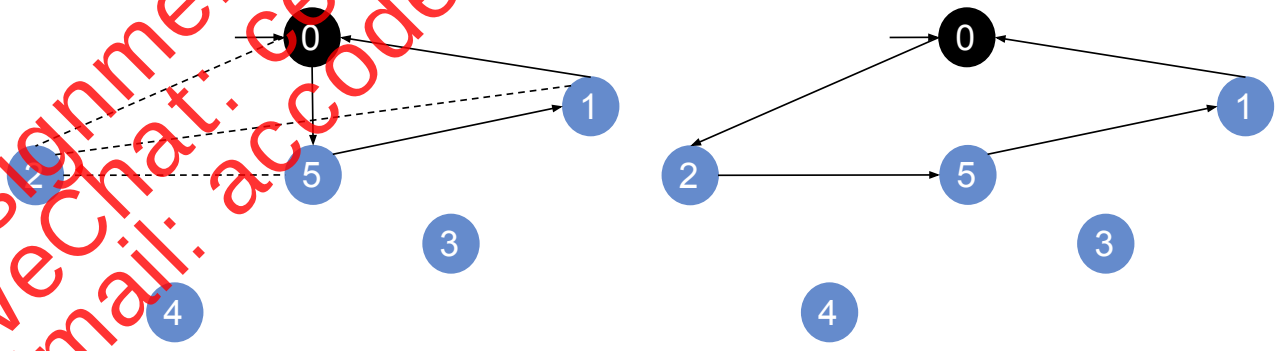


Figure 14: Step 3 of Minmax Insertion



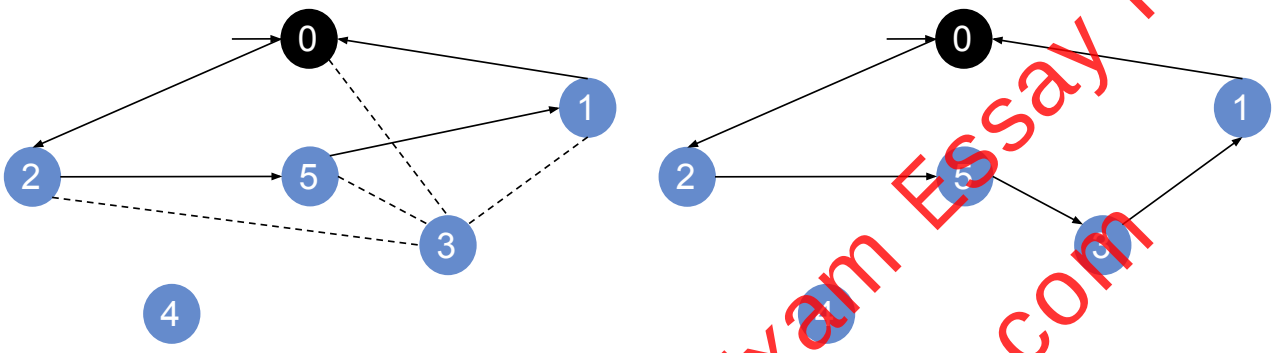
(a) Select the vertex (b) Insert the vertex

Figure 15: Step 4 of Minmax Insertion



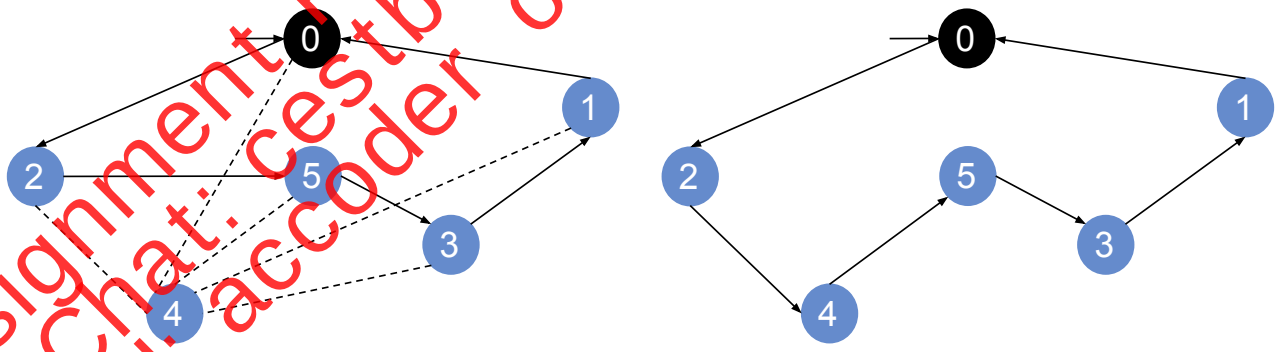
(a) Select the vertex (b) Insert the vertex

Figure 16: Step 5 of Minmax Insertion



(a) Select the vertex (b) Insert the vertex

Figure 17: Step 6 of Minmax Insertion



(a) Select the vertex (b) Insert the vertex

Figure 18: Step 7 of Minmax Insertion

3 Running your programs

Your program should be able to be ran like so:

```
$ ./<program_name>.exe <coordinate_file_name> <output_file_name>
```

Therefore, your program should accept a coordinate file, and an output file as arguments. Note that C considers the first argument as the program executable. Both implementations should read a coordinate file, run either smallest sum insertion or MinMax insertion, and write the tour to the output file.

3.1 Provided Code

You are provided with the file **coordReader.c**, which you will need to include this file when compiling your programs.

1. **readNumOfCoords()**: This function takes a filename as a parameter and returns the number of coordinates in the given file as an integer.
2. **readCoords()**: This function 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.
3. **writeTourToFile()**: This function takes the tour, the tour length, and the output filename as parameters, and writes the tour to the given file.

4 Instructions

- Implement a serial solution for the smallest sum insertion and the MinMax insertion. Name these: ssInsertion.c, mmInsertion.c.
- Implement a parallel solution, using OpenMP,for the smallest sum insertion and the MinMax insertion algorithms. Name these: ompssInsertion.c, ompmmInsertion.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.
  - **make ssi** compiles ssInsertion.c and coordReader.c into ssi.exe with the GNU compiler
  - **make mmi** compiles mmInsertion.c and coordReader.c into mmi.exe with the GNU compiler

  - **make ssomp** compiles ompssInsertion.c and coordReader.c into ssomp.exe with the GNU compiler
  - **make mmomp** compiles ompmmInsertion.c and coordReader.c into mmomp.exe with the GNU compiler
  - **make issomp** compiles ompssInsertion.c and coordReader.c into issomp.exe with the Intel compiler
  - **make immomp** compiles ompmmInsertion.c and coordReader.c into immomp.exe the Intel compiler

- Test each of your parallel solutions using 1, 2, 4, 8, 16, and 32 threads, recording the time it takes to solve each one. 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**.
- Plot a speedup plot with the speedup on the y-axis and the number of threads on the x-axis for each parallel solution.
- Plot a parallel efficiency plot with parallel efficiency on the y-axis and the number of threads on the x-axis for each parallel solution.
- Write a report that, for each solution, **using no more than 1 page per solution**, describes: your serial version, and your parallelisation strategy.
- In your report, include: the speedup and parallel efficiency 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 upload should be:**
  1. **Makefile**
  2. **ssInsertion.c**
  3. **mmInsertion.c**
  4. **ompssInsertion.c**
  5. **ompmmInsertion.c**
  6. **report.pdf**
  7. **The slurm script** you used to run your code on Barkla.

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));
```

For a 2-D array:

```
int **twod_array = (int **)malloc(numOfElements * sizeof(int*));
for (int i = 0; i < numOfElements; i ++){
    twod_array[i] = (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}

ssi: ssInsertion.c coordReader.c
    gcc ssInsertion.c coordReader.c -o ssi.exe -lm
```

Now, on the command line, if you type 'make ssi', 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.

This command may work for you and it may not. The point is to allow you to compile however you like. If you want to declare the iterator in a for loop, you would have to add the compiler flag **–std=c99**. **–fopenmp** is for the GNU compiler and **–qopenmp** is for the Intel Compiler. If you find that the MakeFile is not working, please get in contact as soon as possible.

Contact: h.j.forbes@liverpool.ac.uk

6 Marking scheme

1	Code that compiles without errors or warnings	15%
2	Same numerical results for test cases (tested on CodeGrade)	20%
3	Speedup plot	10%
4	Parallel Efficiency Plot	10%
5	Parallel efficiency up to 32 threads (tests on Barkla yields good efficiency for 1 Rank with 1, 2, 4, 8, 16, 32 OMP threads)	15%
6	Speed of program (tests on Barkla yields good runtime for 1, 2, 4, 8, 16, 32 ranks with 1 OMP thread)	10%
7	Clean code and comments	10%
8	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. This assessment accounts for 40% of your final mark, however as it is a resit you will be capped at 50% unless otherwise stated by the Student Experience Team. 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 23:59 GMT Friday the 2nd of August 2024. https://www.liverpool.ac.uk/aqsd/academic-codes-of-practice/code-of-practice-on-assessment/