In this assignment, you are asked to implement a numerical method for solving a partial differential equation. This document explains the operation in detail, so you do not have to have studied calculus. 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 Laplace solver

Modelling heat transfer in a room can be done by using the Laplace equation, a second-order partial differential equation. This can be approximated using a iterative stencil method. Consider this two-dimensional array which represents a $25m^2$ room.

10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	**100**
10	10	10	10	10	10	10	10	10	**100**
10	10	10	10	10	10	10	10	10	**100**
10	10	10	10	10	10	10	10	10	**100**
10	10	10	10	10	10	10	10	10	**100**
10	10	10	10	10	10	10	10	10	**100**
10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10

Figure 1: An example of the room

This two-dimensional array represents the space in the room, where the dimensions are NxN. Each element in this array represents the temperature of that point within the room. The boundaries of the array represent the walls. The points equal to 100, represent a radiator within the room. The radiator always occupies 60% of the right wall and is centred. That is the radiator starts at **t[N−1][floor((N−1)∗0.3)]**, and ends at **t[N−1][ceil((N−1)∗0.7)]** assuming 0 based indexing. Note that the room will always be $25m^2$. That means the number of the points only changes the resolution of the points in the room, not the actual size of the room.

To model how the heat from the radiator moves throughout the room, we use the following calculation for each point.

curr_t[i][j]=AVERAGE(prev_t[i][j+1]+prev_t[i][j−1]+prev_t[i+1][j]+prev_t[i−1][j])

Figure 2: The iterative calculation to find the temperature moving through the room

That is, each point is equal to the average of the surrounding points. When applying this to **Figure 1**, we have these new temperatures. **Figure 3** is after the first iteration, and **Figure 4** is after the second iteration.

10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	32.5	**100**
10	10	10	10	10	10	10	10	32.5	**100**
10	10	10	10	10	10	10	10	32.5	**100**
10	10	10	10	10	10	10	10	32.5	**100**
10	10	10	10	10	10	10	10	32.5	**100**
10	10	10	10	10	10	10	10	32.5	**100**
10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10

Figure 3: An example of the room after one iteration

10	10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	15.625	10
10	10	10	10	10	10	10	15.625	38.125	**100**
10	10	10	10	10	10	10	15.625	43.75	**100**
10	10	10	10	10	10	10	15.625	43.75	**100**
10	10	10	10	10	10	10	15.625	43.75	**100**
10	10	10	10	10	10	10	15.625	43.75	**100**
10	10	10	10	10	10	10	15.625	38.125	**100**
10	10	10	10	10	10	10	10	15.625	10
10	10	10	10	10	10	10	10	10	10

Figure 4: An example of the room after two iterations

We can observe that we do not update the boundary elements, in order to avoid any access to memory that does not exist. To update the array, only the elements within the range of indices from the second row to the second-to-last row (rows 1 to N-2) and from the second column to the second-to-last column (columns 1 to N-2) should be modified.

1.1 OpenMP laplace solver

You are asked to implement this operation in a C function with the following signature. This function should be saved in a file called heat.c

```
double get_final_temperatures(int N, int maxIter, double radTemp){

        //...your code here

        int pointx = floor((N-1)*0.5);
        int pointy = floor((N-1)*0.5);
        double result = curr_t[pointx][pointy];

        return result;
}
```

N is the number of points along one axis of the room matrix and **maxIter** is the number of iterations to be performed in one run. Both **pointX** and **pointY** are the coordinates for the centre of the room and therefore **curr_t[pointx][pointy]** is the temperature at the centre of the room. **radTemp** is the value for the radiator to be set to. Therefore the function returns the temperature of the centre of the room for a given radiator temperature after **maxIter** iterations have been performed.

1.2 Serial Implementation

You are asked to implement a sequential main C file which can do the following.

- Read a string of radiator temperatures from an input file and store them in a one-dimensional array.
- Call the 'get_final_temperature()' function for each temperature.
- Store the results in a one-dimensional array.
- Write the results to an output file.

Once compiled, the sequential program should be called like so:

```
$ ./<program> <N> <max_iter> <input_file_name> <output_file_name>
```

Where **<program>** is the executable, **<N>** is the size of N (defines the number of points in the room), **<max_iter>** is the number of iterations to be performed for each radiator temperature, and **<output_file_name>** is the name of the output file.

1.3 Distributed implementation

This time, you are asked to implement a distributed main c file which can perform the same functionality of the serial implementation, but distribute the radiator temperatures that are read from the input file between MPI ranks.

Once compiled, the distributed program should be called like so:

```
$ mpirun -np <num_ranks> ./<program> <N> <max_iter>
<input_data_file> <output_data_file>
```

Where **<num_ranks>** is the number of MPI processes. The other arguments given here are the same as those explained for the serial version.

1.4 Data file format

The first line of the input file has an integer. This integer defines the number of temperatures in the file. The second line of the data file is a space-separated list of all the radiator temperature values. The output file will follow the same format. The input files' names follows: input_K.dat where **K** is the number of values.

The output files' names provided follow: output_K_N_maxIter where **K** is number of values, and **N** and **maxIter** are the arguments explained above.

1.5 Provided code

You are provided with the code file-reader.c which contains the following functions.

- **read_num_of_temps**: Which takes an **input file's name** as an argument, and returns the first line of file (the number of radiator temperatures)
- **read_temps**: Which takes the **input file's name** and the **numOfTemps** as arguments, and returns a one-dimensional array of temperatures.
- **write_to_output_file**: Which takes the **output file's name**, the **array of room temperatures found** and the **numOfTemps** as arguments, and writes to an output file.

2 Instructions

- Implement a multi-threaded laplace solver using OpenMP. Save it in a file called **heat.c**.
- Modify the main-serial.c file so that it reads from the input data file, calls your OpenMP stencil function, and writes to the output data file. Use the output to make sure your implementation is correct. Ensure your code is saved as **main-serial.c**.
- Modify the main-mpi.c file so that it performs the same functionality as main-serial.c but distributes the radiator temperatures over multiple MPI processes. Ensure it is saved as **main-mpi.c**.
- Write a Makefile that includes instructions to compile your programs. Your MakeFile should work like so:
  - **make gccserial** - compiles 'main-serial.c', 'heat.c' and 'file-reader.c' into 'heat-omp-gcc' with the GNU compiler (gcc)
  - **make gcccomplete** - compiles 'main-mpi.c', 'heat.c' and 'file-reader.c' into 'heat-complete-gcc' with the GNU mpi compiler (mpicc)
  - **make iccserial** - compiles 'main-serial.c', 'heat.c' and 'file-reader.c' into 'heat-omp-icc' with the Intel compiler (icc)
  - **make icccomplete** - compiles 'main-mpi.c', 'heat.c' and 'file-reader.c' into 'heat-complete-icc' with the Intel mpi compiler (mpiicc)
- Try running your program for 1, 2, 4, 8, 16 and 32 OpenMP threads, measuring the time taken in each instance. Use this to plot a speedup plot with speedup on the y-axis and the number of threads on the x-axis.
- Test the fastest running instance (up to 8 threads) over 1, 2, 4, 8, 16 and 32 ranks i.e. if you found that 4 OpenMP threads was the fastest, test this with 1, 2, 4, 8, 16, 32 ranks. Use this to draw a strong-scaling plot with time on the y-axis and the number of ranks on the x-axis.
  - The maximum number of nodes you will need for 8 OpenMP threads and 32 MPI ranks is 8 nodes.
  - You will potentially have to wait hours/potentially a few days if you submit to multiple nodes. If there is little time until the deadline, test with 1 OpenMP thread up to 32 ranks on the course node.
- **When testing your program on Barkla to get your results for the speedup and strong-scaling plot, ensure you test with N = 256 maxIter = 4096 with the large input file input_1024.dat**

- Using up to one page for each code you produced (not including images), write a report that describes:
  - your implementation and parallel strategy for heat.c, and its speedup plot
  - your implementation for main-serial.c
  - your implementation and parallel strategy for main-mpi.c, and its strong-scaling plot
  - for each plot, how you measured and calculated it, including a table with your times and why your program achieved a linear speedup/reduction in time or not

  Include a screenshot of compiling and running your program, making sure your username is visible.

- **Your final submission should include:**
  1. **heat.c** - the parallel implementation using OpenMP.
  2. **main-serial.c** - a main function that calls the function defined in heat.c to perform the operations described above.
  3. **main-mpi.c** - the complete implementation using OpenMP and MPI.
  4. **Makefile** - a MakeFile that can compile 4 different programs. The instructions for this are given above.
  5. **Report.pdf** - a pdf file containing the plots, descriptions, and screenshots.
  6. **The slurm script** you used to run your code on Barkla.
- **This assignment should be uploaded on Codegrade, following the instructions present there**
- Failure to follow any of the above instructions is likely to lead to reduction in scores.

3 Hints

If you get any segmentation faults when running your program, use a tool called gdb to help debug. Read its manual to understand how to use it.

Make sure to test your code with small as well as big matrices.

Ensure that you are not printing the room temperatures when doing the large test. This will greatly affect your runtime, especially for the large files.

The memory movement of copying curr_t into prev_t at the end of every iteration can have big consequences on the time it takes for the program to run. It would be more efficient if you had a 3-D array t[2][N][N]. You could switch between t[0][N][N] and t[1][N][N] depending which is the current or previous iteration.

If your sequential run for N = 256 maxIter = 4096 with input_1024.dat is taking longer than 10 minutes, reconsider your strategy.

3.1 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}

gccserial: heat.c main-serial.c file-reader.c
    gcc -fopenmp heat.c main-serial.c file-reader.c -o
    heat-omp-gcc -lm
```

Now, on the command line, if you type 'make gccserial', 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

4 Marking scheme

1	Code that compiles without errors or warnings	5%
2	Same numerical results for test cases (tested on CodeGrade)	30%
3	Speedup plot (clean plot, correct axes and explained)	10%
4	Strong scaling plot (clean plot, correct axes and explained)	10%
5	Scaling efficiency up to 32 threads (tests on Barkla yields good scaling efficiency for 1 Rank with 1, 2, 4, 8, 16, 32 OMP threads)	10%
6	Scaling efficiency up to 32 ranks (tests on Barkla yields good scaling efficiency for 1, 2, 4, 8, 16, 32 ranks with 1 OMP thread)	10%
7	Clean code and comments (clean, well structured readable cde)	10%
8	Report (explained parallel strategy and implementation for each code produced, i.e. why certain decisions were made to gain performance such as why certain OMP directives/MPI calls used)	15%

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 accounts for 35% 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.

5 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/