In this assignment, you are asked to implement a linear algebra operation that is at the heart of many computations we do. You do not need to have previously studied linear algebra since this document explains the operation in detail. You are encouraged to use your spare time in your labs to work on this assignment. The brief may seem intimidating, but the majority of it is defining how exactly you should do this assignment and defining the code you are provided and providing hints.

1 Two-Dimensional Box Stencils

$$\text{Input image}_{m,n} = \begin{bmatrix} \mathbf{7} & \mathbf{2} & \mathbf{3} & 3 & 8 \\ \mathbf{4} & \mathbf{5} & \mathbf{3} & 8 & 4 \\ \mathbf{3} & \mathbf{3} & \mathbf{2} & 8 & 4 \\ 2 & 8 & 7 & 2 & 7 \\ 5 & 4 & 4 & 5 & 4 \end{bmatrix} \text{Filter}_{k,k} = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

$$\text{Output image}_{m,n} = \begin{bmatrix} 7 & 2 & 3 & 3 & 8 \\ 4 & \mathbf{6} & -9 & -8 & 4 \\ 3 & -3 & -2 & -3 & 4 \\ 2 & -3 & 0 & -2 & 7 \\ 5 & 4 & 4 & 5 & 4 \end{bmatrix}$$

This operation takes as input two matrices (two-dimensional arrays). We call one of them the “input image” and the other the “filter”. The input image can be assumed to be at least the same size as the filter in both dimensions, if not larger. The filter can be assumed to always be square, while the input image need not be a square matrix (i.e. its two dimensions are of a different size). The operation produces as output, an “output image” that is the same size as the input. The input and output images are of dimensions $m \times n$ while the filter is of dimensions $k \times k$, and we can assume $m >= k, n >= k$.

The operation is applied point-wise to the input image. To start, we identify the centre point of the filter. This is element (k_m, k_m), where $k_m = floor((k-1)/2)$. To calculate element (i, j) of the output matrix, the filter is centred on the point (i, j) of the input matrix, and the corresponding elements of the input matrix are multiplied by the elements of the filter matrix. These products are then all summed to get the value of the output matrix at point (i, j). For the example in Figure 1, the filter is centred on point (1, 1), which has the value 5.

To calculate the value of the output element (1, 1), we multiply all corresponding elements of the input matrix and the filter matrix:

7 * 1 + 2 * 0 + 3 * (-1) + 4 * 1 + 5 * 0 + 3 * (-1) + 3 * 1 + 3 * 0 + 2 * (-1) = 6.

Similarly, for element (1, 2) of the output matrix, the value is calculated as:

2 * 1 + 3 * 0 + 3 * (-1) + 5 * 1 + 3 * 0 + 8 * (-1) + 3 * 1 + 2 * 0 + 8 * (-1) = -9.

Note that we would never want to read outside the bounds of the input array, nor pad the input array with zeros. Therefore, this operation will leave a "boundary region" where the values of the output matrix are not yet defined. For example, while the input has rows from 1 to m, the filter would start to be applied at row $0 + b_{lower}$ and end at row $m - 1 - b_{upper}$. We will follow $b_{lower} = floor(k-1)/2$ and $b_{upper} = ceiling(k-1)/2$. For the elements in this boundary region at the beginning and end of each dimension, we copy the corresponding elements from the input image to the output.

2 Tasks

2.1 Task 1: OpenMP Stencil Solver

You are asked to implement a multi-threaded version of the operation in Section 1 in a C function, using OpenMP. This function should have the following signature.

```
void stencil(float *input_vec, float *output_vec,
             float *filter_vec, int m, int n, int k, int b){

        float (*input)[m][n] = (float(*)[m][n]) input_vec;
        float (*filter)[k] = (float(*)[k]) filter_vec;
        float (*output)[m][n] = (float(*)[m][n]) output_vec;

        //Your code starts here

}
```

Here, ∗input_vec, ∗ filter_vec , and ∗output_vec are pre-allocated one-dimensional arrays of the appropriate size, being passed as pointers to the first element. The provided lines of code change the 1-D arrays. The lines convert the input and output arrays to 3-D arrays, so that b matrices of size $m \times n$ are processed in a single function call, while the filter matrix is changed to 2-D as described above. In other words, your code is expected to perform this operation in batches of size b. After the given three lines of code,this function can treat input and output as 3-D arrays of size [b][m][n], and kernel as a 2-D array of size [k][k]. e.g. the following line is valid: output [0][0][0] = input[0][0][0] * filter[0][0];

2.2 Task 2: Nearly-serial implementation

You are asked to write a nearly-serial program that can do the following:

1. Read two files. An input file containing b matrices of size $m \times n$, and a filter file containing a filter of size $k \times k$.
2. Call your OpenMP stencil function and write the results to an output array.
3. Write your output array to an output file.

Once compiled, your program should be able to be run like so:

```
$ ./program <input_file> <filter_file> <output_file>
```

Where 'input_file' contains the data for the input image, 'filter_file' contains the data for the filter, and 'output_file' is the file **your program is expected to write the output array to**. The structure of the files is discussed in section 3.1.

2.3 Task 3: Distributed implementation

Finally, you are asked to write a 'complete solution' using MPI that can perform the same operation as the nearly-serial solution, but distribute the matrices across multiple processes.

Your program should be run with the following command:

```
$ mpirun -np <num_procs> ./program <input_file> <filter_file>
<output_file>
```

Where 'num_procs' is the number of processes that will run your program, and the other variables are the same for the 'Nearly-serial' implementation.

3 Given materials

For this assignment, you are given some data files, and are provided the code to read them.

3.1 Data file format

The first line of each file has some integers, each followed by a space. These integers define the dimensionality and map onto b, m, and n.

The second line of the data files is a space-separated list of all the values in the array flattened out in a row-major one-dimensional format, each with 7 digits after the decimal.

The name of the files follows as such:

- input_b_m_n.dat - The input image file, where:
  - b is the number of matrices,
  - m is the size of the matrices' m dimension,
  - n is the size of the matrices' n dimension.
- filter_k.dat: The filter file, where k is the size of the filter.
- output_b_m_n_k.dat: The output file containing the correct output for the corresponding input file (b, m, n) and the filter file (k). Note that you do not have to call your output file this, you can call it anything. These files just provide the solutions.

These files are available on Barkla in the directory **/mnt/data1/users/forbes/comp328_io**. You must run the provided Bash script **CA1_setup.sh** to set up a directory with symbolic links to the provided files. **Do not manually copy the files, especially the large files, as it will use unnecessary disk space on Barkla.** It is recommended that as soon as you confirm your output is correct for a large data file, that you delete the output to manage space.

3.2 Provided code

You are given the code file-reader.c which has the following functions:

- **read_num_dims**: Takes a filename as a parameter and returns the number of dimensions in the file as an integer.
- **read_dims**: Takes a filename and the number of dimensions as parameters, and returns an array of dimensions. For the input file, the array is in the order b, m and n, whereas the filter is always square.
- **read_array**: Takes as parameters: a filename and the array of dimensions and the number of dimensions. This function returns a one-dimensional array of floats that are read from the given file.
- **write_to_output_file**: Takes as parameters: a filename, a one-dimensional output array, the array of dimensions and the number of dimensions. This function writes dimensions, and the output array to the given file.
- **product**: Takes the array of dimensions as an and the number of dimensions as a parameter, and returns the product of them. Used to find the total number of elements given the dimensions.

4 Instructions

- Implement a multi-threaded solver for the stencil problem in section 1 using OpenMP. Save it in a file called **stencil.c**.
- Implement a nearly-serial solution that reads the input and filter files, calls your stencil function, and writes to the output file. Call this **main-nearly.c**
- Implement a complete parallel solution that performs the same task as main-nearly.c, but distributes the input matrices over multiple MPI processes. Call this **main-mpi.c**
- Write a Makefile that includes instructions to compile your programs. Your MakeFile should work like so:
  - **make gccnearly** - compiles 'main-nearly.c', 'stencil.c' and 'file-reader.c' into 'stencil-nearly-gcc' with the GNU compiler (gcc)
  - **make gcccomplete** - compiles 'main-mpi.c', 'stencil.c' and 'file-reader.c' into 'stencil-complete-gcc' with the GNU MPI compiler (mpicc)
  - **make iccnearly** - compiles 'main-nearly.c', 'stencil.c' and 'file-reader.c' into 'stencil-nearly-icc' with the Intel compiler (icc)
  - **make icccomplete** - compiles 'main-mpi.c', 'stencil.c' and 'file-reader.c' into 'stencil-complete-icc' with the Intel compiler (mpiicc)
- Run your nearly-serial, or distributed solution with 1 MPI process, for 1, 2, 4, 8, 16 and 32 OpenMP threads, with the datafiles **input_1_3840_2048.dat** and **filter_5.dat**, i.e. strong scaling.
- Test your fastest running instance of OpenMP threads (max 8 threads) with 1, 2, 4, 8, 16 and 32 MPI processes i.e. if you found that 4 OpenMP threads was the fastest, test this with 1, 2, 4, 8, 16, 32 MPI processes. When running these tests, ensure that the data files used contain as many matrices as the number of MPI processes i.e. weak scaling.
  - You will need to submit to multiple nodes for this.
  - Check the hints section or Lecture 08 on how to submit to run a job over multiple nodes.
  - The maximum number of nodes you will need for 8 OpenMP threads and 32 MPI ranks is 8 nodes.
- **When timing your programs, ensure you only time the stencil function and not the whole program.** At the end of your programs, in order extract your stencil function's time, include this line:

```
printf("STENCIL_TIME: %f", end_time - start_time);
```

Where end_time is the wall clock time when the stencil function completes, and start_time is the wall clock time when your stencil function starts.

When running your MPI version, ensure it is only rank 0 that prints the time.

- Write a report containing:
  - an explanation of each OpenMP directive and MPI routine you used and a justification of why you have used them (1 page maximum)
  - (optional) any techniques you used that you want to highlight
  - a table containing the runtime, speedup, and parallel efficiency for each number of OpenMP threads used for the nearly-serial implementation (or distributed implementation with 1 MPI process)
  - a table containing the runtime for each MPI process used for the distributed implementation
  - a speedup plot for OpenMP threads for the nearly-serial implementation (or distributed implementation with 1 MPI process) with speedup on the y-axis and number of OpenMP threads on the x-axis. Include an explanation, with reference to your code and theory, about whether you were able to achieve linear speedup or not. (max 1 page)
  - a weak-scaling plot with time on the y-axis and number of MPI processes on the x-axis. Include an explanation, with reference to the your code and theory, about why you were able to achieve ideal scaling or not. Also, ensure sure you specify how many OpenMP threads you used for the results. (max 1 page)
  - a screenshot of you compiling your program with your username visible.
- **Your final submission should include:**
  1. **stencil.c** - the OpenMP implementation of the stencil problem.
  2. **main-nearly.c** - the nearly-serial implementation.
  3. **main-mpi.c** - the distributed implementation using 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 marks.

5 Hints

5.1 General 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 you use dynamic memory allocation (malloc) when declaring large arrays. nb
- Make sure to test your code with small as well as big input files. **DO NOT TEST THE LARGE DATAFILES ON THE LOGIN NODES ON BARKLA**
- If your code works on Barkla but not CodeGrade or vice versa **message Henry immediately**
- Ensure when you are performing your tests, that you are not printing any debugging information as printing will be expensive if you're doing it over thousands of iterations
- If your serial run is taking longer than 5 minutes, profile your code and rethink your strategy.
- Ensure you are taking advantage of vectorisation where possible. With the Intel compiler, use -qopt-reportN as a compiler flag, to generate optimisation reports to know where your code is being optimised and where it isn't.
- Try both the Intel Compiler and the GNU Compiler. Don't forget to use optimisation flags!
- For running on multiple nodes, have a look at Lab04. If you are close to the deadline and have still not tested your MPI implementation across multiple nodes, test on one node with 1 OpenMP thread for all the runs.
- Because of the size of the datafiles, you may find that reading the files takes longer than executing the stencil function. That is why you are asked to time only the stencil function.

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

6 Marking scheme

A	**Code**	**40%**
A1	Code that compiles without errors or warnings (tested on CodeGrade)	5%
A2	Same numerical results for test cases (tested on CodeGrade)	30%
A3	Clean code and comments (clean, well structured and readable code)	5%
B	**Performance of Code**	**25%**
B1	Speed of stencil (tests on Barkla show good speed taken from the fastest running instance across 1, 2, 4, 8, 16, and 32 threads)	12.5%
B2	Scaling efficiency up to 32 ranks (tests on Barkla yields good scaling efficiency for 1, 2, 4, 8, 16 and 32 ranks)	12.5%
C	**Report**	**35%**
C1	Explaining what OpenMP and MPI routines you used, and explain how and why you used them	10%
C2	Results table for OpenMP with: number of threads, timings, speedup and parallel efficiency for each OpenMP thread, with a speedup plot (showing speedup on the y-axis, the number of OpenMP threads on the x-axis with each point clearly marked)	5%
C3	Results table for MPI with: number of processes, timings, and number of OpenMP threads used, with a weak-scaling plot (showing time on the y-axis, the number of MPI processes on the x-axis, with each point clearly marked.)	5%
C4	Detailed explanation for why code achieved linear scaling or not for both versions.	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 20% of your final mark. 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 17:00 GMT Friday the 2nd of May 2025. https://www.liverpool.ac.uk/aqsd/academic-codes-of-practice/code-of-practice-on-assessment/