CITS3402/CITS5507: Assignment 2

Semester 2, 2025

Assessed, worth **15%**. **Due 11:59 pm Friday 10th October**.

1 Outline

Your task is to extend the previous OpenMP convolution assignment by producing a fast **distributed-memory** implementation of 2D convolution using the Message Passing Interface (MPI). In addition to computing the convolution, your program must support *stride* in both the vertical and horizontal directions. **The project can be done individually or in a group of two.**

While this assignment is centred around programming, your submission is heavily weighted towards the report. Do not expect to receive full marks for an implementation without thorough analysis.

2 Description

The second project is a continuation of the first project. The problem is the same: *implementation of 2D convolution.*

In the first project, the task was to perform the convolution using threads, using a shared memory model. In this second project, you will use both distributed memory and shared memory computation.

As before, the 2D discrete Convolution of an input f and kernel g is

$$(f * g)[n, m] = \sum_{i=-M}^{M} \sum_{j=-N}^{N} f[n+i, m+j]g[i, j].$$

For this assignment we will again use padding and always use "same" padding (see the example from previous assignment in Figure 1).

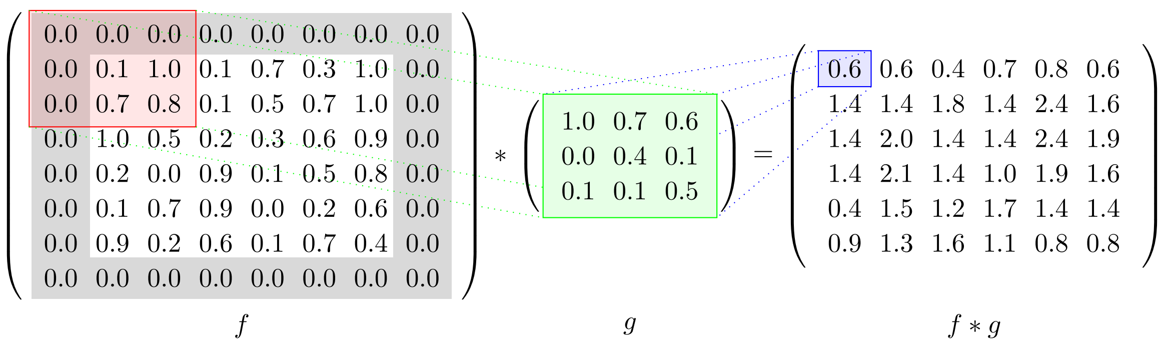
$$\underbrace{\begin{pmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.1 & 1.0 & 0.1 & 0.7 & 0.3 & 1.0 & 0.0 \\ 0.0 & 0.7 & 0.8 & 0.1 & 0.5 & 0.7 & 1.0 & 0.0 \\ 0.0 & 1.0 & 0.5 & 0.2 & 0.3 & 0.6 & 0.9 & 0.0 \\ 0.0 & 0.2 & 0.0 & 0.9 & 0.1 & 0.5 & 0.8 & 0.0 \\ 0.0 & 0.1 & 0.7 & 0.9 & 0.0 & 0.2 & 0.6 & 0.0 \\ 0.0 & 0.9 & 0.2 & 0.6 & 0.1 & 0.7 & 0.4 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}}_{f} * \underbrace{\begin{pmatrix} 1.0 & 0.7 & 0.6 \\ 0.0 & 0.4 & 0.1 \\ 0.1 & 0.1 & 0.5 \end{pmatrix}}_{g} = \underbrace{\begin{pmatrix} 0.6 & 0.6 & 0.4 & 0.7 & 0.8 & 0.6 \\ 1.4 & 1.4 & 1.8 & 1.4 & 2.4 & 1.6 \\ 1.4 & 2.0 & 1.4 & 1.4 & 2.4 & 1.9 \\ 1.4 & 2.1 & 1.4 & 1.0 & 1.9 & 1.6 \\ 0.4 & 1.5 & 1.2 & 1.7 & 1.4 & 1.4 \\ 0.9 & 1.3 & 1.6 & 1.1 & 0.8 & 0.8 \end{pmatrix}}_{f * g}$$

Figure 1: Example of 2D convolution operation with "same" padding.

Along with the "same" padding from assignment 1, you will also now incorporate a **stride** parameter $s_H \times s_W$, where the kernel is moved by s_H rows and s_W columns between successive output locations. When $s_H, s_W > 1$ the output has reduced resolution. See Figure 2 for a 2D convolution example with stride of 2 in the horizontal ($s_W = 2$) and 3 in the vertical ($s_H = 3$).

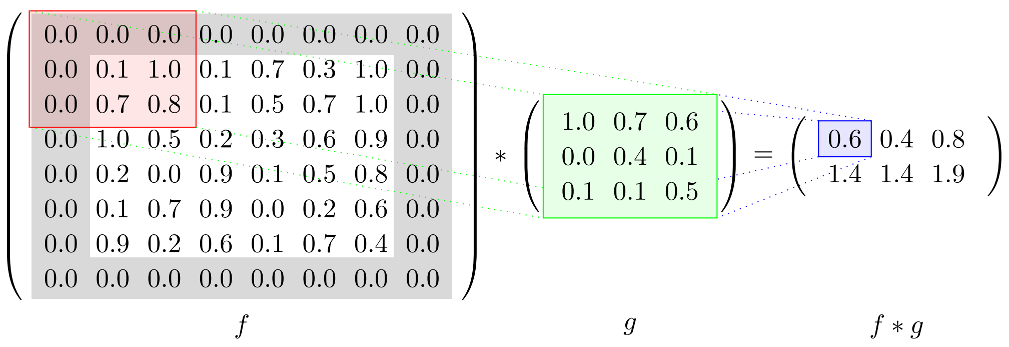

Figure 2: Example of strided 2D convolution operation with $s_W = 2$ and $s_H = 3$.

Your task for Assignment 2 is to produce a fast parallel implementation of 2D convolution using both distributed memory (MPI) and shared memory computation OpenMP.

3 Tasks

1. Implement both a single-threaded and parallel version of 2D convolution, including padding (as was the case for Assignment 1), and incorporate potential stride parameters. Your implementation must work on single-precision floating-points (float32s), and be able to handle arbitrarily large arrays. You should (at least) investigate sequential, OpenMP, MPI and OpenMP+MPI performance.

   Your functions should have the following basic definition:

```
void conv2d_stride(
    float **f,       // input feature map (padded)
    int H, int W,    // global input size
    float **g,       // kernel
    int kH, int kW,  // kernel size
    int sH, int sW,  // stride in height and width
    float **output,  // local output
    MPI_Comm comm    // communicator
);
```

2. Implement functions for creating and reading/writing inputs (f), kernels (g), and outputs $f * g$. Your submission should include a main file that can:

   - Generate a random input of size $H \times W$, kernel of size $kH \times kW$ and stride $s_H \times s_W$. Optionally write the input/output to a file (see Example 7.2.1).
   - Support arbitrary stride values (both vertical and horizontal) for the convolution, ensuring the output size and indexing correctly reflect the chosen stride.

   You may wish to use getopt to do this. Don't forget to document how to run your code in your report.

3. Ensure that your implementation produces correct results. A number of examples have been uploaded to the LMS so you can test your code. NOTE: Passing these test cases does not mean your solution is correct. You need to do thorough testing on your own.

4. Time the performance of your code. Do not include I/O (or random matrix generation) in the timing of your implementation.

5. Write a report describing your parallel implementation and its performance. Your code should include (at least) the following points:

- Description of how you have parallelised your algorithm. This explanation should reference both distributed memory and shared memory computation.
- Description of how you are representing your arrays (f, g, $f * g$) in memory and communicating between processes.
- Cache and memory-layout considerations within each MPI process.
- The effect of stride on computational and communication cost.
- Thorough performance metrics and analysis of your solution describing the benefits and speedup of parallelism with respect to the number of threads and processes.

4 HPC resources

While you should do basic development and debugging locally, we expect you to use both Kaya and Setonix for analysing the execution time of your solution for your report.

You should stress test your program, and see what the largest inputs you can handle in under 15 minutes of time on up to four nodes on Kaya and Setonix. When running on Setonix please note that we have a limited number of CPU hours.

5 Submission

You should submit your assignment via LMS. The submission must include your source code (including a Makefile), and your report as a PDF in a **zip/tar file**. Please also include a **README** on how to run your code. All files must have the names and student numbers of both group members.

If we cannot build your project by simply running `make` on Kaya and Setonix, you may receive zero for correctness.

6 Marking Rubric

Criteria	Proficient (100%)	Competent (66%)	Novice (33%)	Marks
Implementing serial convolution with stride	Correct implementation of stride. Code was not messy, well documented, and ran successfully.	Correct implementation of stride, but code was hard to follow.	Implementation of stride not correct with major issues.	1
Matrix generation and I/O, main file	Correct implementation fit to specifications. Code was not messy, well documented, and ran successfully.	Correct implementation fit to specifications, but code was hard to follow.	Code required minor modifications to run, or did not follow specifications correctly.	1
Implementing parallel convolution: distributed memory	Correct implementation with excellent use of MPI for parallelism.	Correct implementation with basic MPI parallelism.	Implementation contained minor mistakes or did not make effective use of MPI.	2
Implementing parallel convolution: distributed memory & shared memory	Correct implementation with excellent use of OpenMP and MPI for parallelism.	Correct implementation with basic OpenMP and MPI parallelism.	Implementation contained minor mistakes or did not make effective use of OpenMP and MPI.	3
Description of parallelism	Thorough analysis with excellent detail.	Good analysis, but lacking minor details.	Brief analysis, significantly lacking details.	5
Description of data decomposition and distribution	Thorough analysis with excellent detail.	Good analysis, but lacking minor details.	Brief analysis, significantly lacking details.	2
Description of communication strategy and synchronisation	Thorough analysis with excellent detail.	Good analysis, but lacking minor details.	Brief analysis, significantly lacking details.	3
Performance metrics and analysis	Thorough analysis with excellent detail.	Good analysis, but lacking minor details.	Brief analysis, significantly lacking details.	10
Formatting and general presentation	Well presented, easy to follow.	Okay presentation, but minor issues.	Poorly presented, hard to read or follow.	3
Total				30

7 Examples

7.1 File specification

An array can be stored in a space-separated text file, where the first row specifies the height and the width of the array. For example the array,

$$\begin{bmatrix} 0.884 & 0.915 & 0.259 & 0.937 \\ 0.189 & 0.448 & 0.337 & 0.033 \\ 0.122 & 0.169 & 0.316 & 0.111 \end{bmatrix} \tag{1}$$

would be represented via the text file:

```
3 4
0.884 0.915 0.259 0.937
0.189 0.448 0.337 0.033
0.122 0.169 0.316 0.111
```

This specification should again be followed throughout this project.

7.2 Program output examples

7.2.1 Generating arrays

Generating arrays, without saving any outputs:

```
$ ./conv_stride_test -H 1000 -W 1000 -kH 3 -kW 3 -sW 2 -sH 3
```

Generating arrays, saving all input and output:

```
$ ./conv_stride_test -H 1000 -W 1000 -kH 3 -kW 3 -sW 2 -sH 3 -f f.txt -g g.txt -o o.txt
```