# CSC2002S Tutorial 2019

## *Parallel Programming with the Java Fork/Join framework: Cloud Classification*

Weather simulation for the purposes of prediction is a complex task that often needs to be completed within a specific time limit in order to be useful. It is therefore a prime candidate for acceleration using parallel programming. Indeed, a significant proportion of the computation budget of supercomputers around the world is devoted to this task.

In this assignment, you will take the output of a simple weather simulation and perform an analysis to determine the prevailing wind direction and the types of clouds that might result. Figure 1 shows a snapshot of a weather simulation with some of the data you will be using.

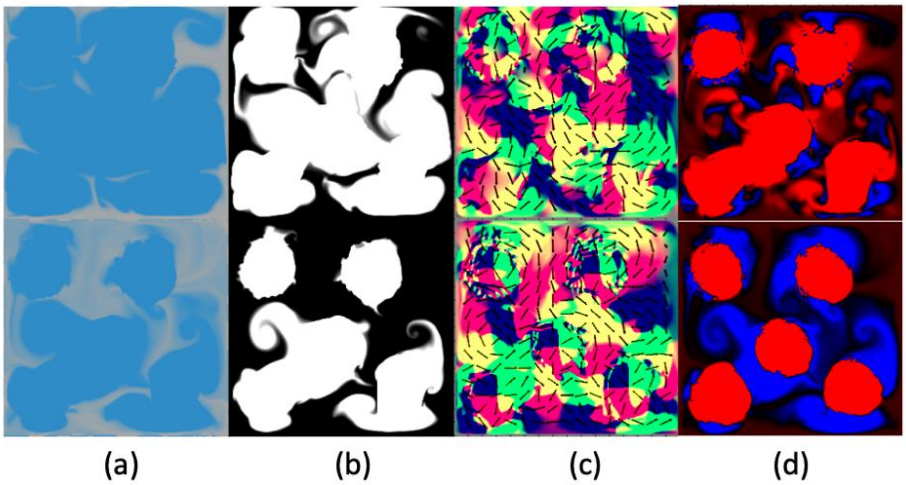



*Figure 1: A snapshot of a weather simulation in progress. This has two layers of different data, including water vapour (a); clouds (b), wind vectors with colouring based on direction (c) and lift values, where uplift is coloured red and sinking blue (d). You will be focusing on the wind and lift values for a single layer over time.*

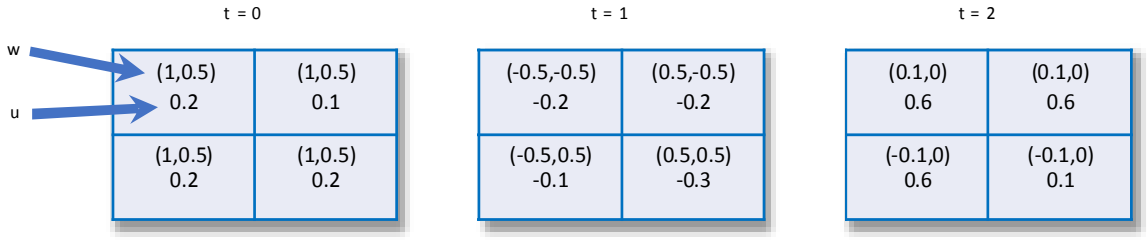
You will be provided with data for a single layer of air as it evolves over time. This air layer at a particular time is represented as a regular two-dimensional matrix with each matrix entry consisting of three floating point values. The first two elements represent the x- and y-coordinate components of a wind vector (w) and the third element is a lift value (u) representing the rate at which air shifts upwards or downwards. These terms are sometimes also called advection (w) and convection (u). Each time step will have a new matrix with changing data.

The file input format is as follows:
<number of time steps – INT><air layer x size – INT> <air layer y size – INT>
<x-coordinate of wind at time = 0, grid pos (0,0) - FLOAT> <y-coordinate of wind at time = 0, grid pos (0,0) - FLOAT> <uplift value at time = 0, grid pos(0,0) - FLOAT> <x-coordinate of wind at time = 0, grid pos (0,1) - FLOAT> … etc.

For example, an unrealistically small 2 x 2 air layer with 3 timesteps might be encoded in the input file as:
3 2 2
1.0 0.5 0.2 1.0 0.5 0.1 1.0 0.5 0.2 1.0 0.5 0.2
-0.5 -0.5 -0.2 0.5 -0.5 -0.2 -0.5 0.5 -0.1 0.5 0.5 -0.3

0.1 0.0 0.6 0.1 0.0 0.6 -0.1 0.0 0.6 -0.1 0.0 0.1

Visually this corresponds to:




Your task is firstly to calculate the prevailing wind over the simulation by finding the average wind vector for all air layer elements and time steps. In the example above the answer truncated to 3 decimal places would be $w_{avg}$ = (0.333, 0.166).

Secondly, assign to each air layer element an integer code (0, 1 or 2) that indicates the type of cloud that is likely to form in that location based on a comparison of the local average wind direction and uplift value.

The local average wind direction is the mean wind of the element and its 8 immediate neighbours. Thus if the element is at index (i,j) the average would combine $w_{(i-1,j-1)}$, $w_{(i-1,j)}$, $w_{(i-1,j+1)}$, $w_{(i,j-1)}$, $w_{(i,j)}$, $w_{(i,j+1)}$, $w_{(i+1,j-1)}$, $w_{(i+1,j)}$, and $w_{(i+1,j+1)}$. At the boundaries you simply exclude elements that would be outside the grid from the calculation of the mean. Thus, if you are on the top edge of the grid (i=0), you would ignore all elements above (the i-1 elements).

Finally, the codes are calculated as follows:

- Code 0 (cumulus) – if the absolute uplift value is greater than the wind magnitude then cumulus clouds are more likely to form (|u| > len(w)). This is because convection provides the blobby shape characteristic of cumulus clouds.
- Code 1 (striated stratus) – When the wind magnitude is greater than a threshold of 0.2 (and also greater or equal to absolute uplift) then stratus cloud may form and be advected in the wind direction.
- Code 2 (amorphous stratus) – Otherwise, nebulous ill-defined stratus clouds are likely.

The file output format is as follows:
<number of time steps – INT><air layer x size – INT> <air layer y size – INT>
<avg. wind x-coordinate – FLOAT><avg. wind y-coordinate – FLOAT>
<cloud classification at time t = 0, grid pos (0,0)> <cloud classification at time t = 1, grid pos (0,1)> …

For the toy problem above this would mean an output of:
3 2 2
0.333333 0.166667
1 1 1 1
0 0 0 0
2 2 2 2

Furthermore, in this assignment you will attempt to parallelize the problem in order to speed it up. You will be required to:

- Use the Java Fork/Join framework to parallelize the sunlight exposure calculations using a divide-and-conquer algorithm.

- Evaluate your program experimentally:

    - Using high-precision timing to evaluate the run times of your serial and your parallel method, across a range of input sizes, and

    - Experimenting with different parameters to establish the limits at which sequential processing should begin.

- Write a report that lists and explains your findings.

Note that parallel programs need to be both **correct** and **faster** than the serial versions. Therefore, you need to demonstrate both correctness and speedup for your assignment.

# Assignment details and requirements

Your program will calculate the sunlight exposure of individual trees in parallel as well as the average sunlight exposure of all trees in the forest.

## 1.1 Input and Output

Your program must take the following command-line parameters:
<data file name> <output file name>
The format of these files must follow the specification provided above.

We will provide you with sample input files (on Vula), though you may also create your own using random data.

## 1.2 Benchmarking

You must time the execution of your parallel sorts across a range of data **sizes** and **number of threads** (sequential cutoffs) and report the speedup relative to a serial implementation. The code should be tested on at least **two different machine architectures** (at least one of which must be a multi-CPU/multi-core machine).

Timing should be done at least 5 times. Use the `System.currentTimeMillis` method to do the measurements. Timing must be restricted to the sunlight exposure code and must exclude the time to read in the files and other unnecessary operations, as discussed in class. Call `System.gc()` to minimize the likelihood that the garbage collector will run during your execution (but do not call this within your timing block!).

## 1.3 Report

You must submit an assignment report **in pdf format**. Your clear and **concise** report should contain the following:

- An *Introduction*, comprising a short description of the aim of the project, the parallel algorithms and their expected speedup/performance.
- A *Methods* section, giving a description of your approach to the solution, with details on the parallelization. This section must explain how you validated your algorithm (showed that it was correct), as well as how you timed your algorithms with different input, how you measured speedup, the machine architectures you tested the code on and interesting problems/difficulties you encountered.
- A *Results and Discussion* section, demonstrating the effect of data sizes and numbers of threads and different architectures on parallel speedup. This section should **include speedup graphs** and **a discussion**. Graphs should be clear and labelled (title and axes). In the discussion, we expect you to address the following questions:
    - Is it worth using parallelization (multithreading) to tackle this problem in Java?
    - For what range of data set sizes does your parallel program perform well?
    - What is the maximum speedup obtainable with your parallel approach? How close is this speedup to the ideal expected?
    - What is an optimal sequential cutoff for this problem? (Note that the optimal sequential cutoff can vary based on dataset size and architecture.)
    - What is the optimal number of threads on each architecture?
- A *Conclusions* (note the plural) section listing the conclusions that you have drawn from this project. What do your results tell you and how significant or reliable are they?

Please do NOT ask the lecturer for the recommended numbers of pages for this report. Say what you need to say: no more, no less.

## 1.4 Assignment submission requirements

- You will need to create and submit a GIT archive (see the instructions on Vula on how to do this).
- Your submission archive must consist of BOTH a technical report and your solution code and a **Makefile** for compilation.
- **Label** your assignment archive with your **student number** and the **assignment number e.g. KTTMIC004_CSC2002S_Assignment1.**
- Upload the file and **then check that it is uploaded.** It is your responsibility to check that the uploaded file is correct, as mistakes cannot be corrected after the due date.
- The usual late penalties of 10% a day (or part thereof) apply to this assignment.

---

Due date:

9am on 9th September 2019

---

- **Late** submissions will be **penalized at 10%** (of the total mark) per day, or part thereof.
- The deadline for marking **queries** on your assignment is **one week after the return of your mark.** After this time, you may not query your mark.

## 1.5 Extensions to the assignment

If you finish your assignment with time to spare, you can attempt one of the following for extra credit:

- Compare the performance of the Fork/Join library with standard threads.
- Replace the square canopy with a circular canopy and measure the impact that this has on performance.
- Not all trees absorb sunlight with the same efficiency. Add a per tree weighting factor for the sunlight exposure. Again, see what impact this has on speedup.

## 1.6 Assignment marking

Your mark will be based primarily on your analysis and investigation, as detailed in the report.

| **Rough/General Rubric for Marking of Assignment 1** | |
|---|---|
| Item | Marks |
| Code – conforms to specification, code correctness, timing, style and comments, produces correct outputs | 15 |
| Documentation - all aspects required in the assignment brief are covered, e.g. Introduction, Methods, Results (with graphs and analysis), Conclusions. | 25 |
| Going the extra mile - excellence/thoroughness/extra work - e.g. additional investigations, depth of analysis, etc. | 5 |
| GIT usage log | 2 |
| Makefile – compile, docs, clean targets | 3 |
| Total | 50 |

Note: submitted code that does not run or does not pass standard test cases will result in a mark of zero. **Any plagiarism, academic dishonesty, or falsifying of results reported will get a mark of 0 and be submitted to the university court**.