CSC2002S

Practical Assignment 3

Parallel Programming with the Java Fork/Join framework:

Cloud Classification

Niceta Nduku

NDKNIC001

13 September 2019

Introduction

The aim of this experiment is to test the performance and compare code that generates cloud information sequentially and in parallel. The cloud data to be performed on is in form of a 3D array with ranging dimensions. The x and y are the air layers and t is the time step for each layer. Each index in the array contains coordinates of wind (advection) and the uplift (convection) at specific time steps. The program must calculate the prevailing wind which is an average of all the wind vectors and classify each wind coordinate based on its local average and convection.

The parallel algorithm uses Java's fork/join framework. The data in the array is divided into smaller sizes until there is a small amount of data to be processed, each being run as a separate thread. The experiment only compares the run times for calculating the prevailing wind and classification. It is expected that the parallel program will run faster than the sequential.

This report looks at how the algorithms works, and benchmarking with varying data and sequential cut offs. The results are then analysed and a conclusion is drawn.

Instructions on how to run the program, are in the readme.txt file that is in the same directory as this document.

Method

Sequential

To run the sequential expreiment, the class SeqCloudData is run in the terminal which begins by taking in two strings. An input file that contains the data to be transformed and an output file where the data will be written to. readData in CloudData class will place all the required data (advection, convection) into their respective 3D arrays. It also creates the 3D array for the cloud classification. The class CloudData was provided by Prof James Bain.

After all the data is put into the respective arrays, the timer will be started by ivoking the method tic() that takes the current time. The method prevailing() that is within SeqCloudData will be called which sums up all the wind vectors with advection and gets the average. Locate(i,ind) is used here to transform the 3D array into a linear array. This was done to ensure that the method used in the parallel is similar. After the prevailing wind has been calculated, the timer will stop by invoking tock(). Tick and tock are methods borrowed from Prof James Bain.

The timer is restarted and the method classification() is called. This method calculates the local average of each wind vector element in advection by calling the method localWindDirection() which takes in the exact coordinates and finds the local average of the wind vector at that cordinate and returns the magnitude of that vector. The magnitude is then compared with the absolute value of the convection at that same coordinate in order to determine the classification. After classification of each wind vector is complete, the timer is stopped.

The data is then written to a file by calling writeData() in CloudData and the times for prevailing wind and classification printed out.

Parallel

The parallel method for the experiment is similar. However, the class ParallelCloudData creates two ForkJoinPool objects for prevailing wind and for classification. When prevailing is called, the prevailing forkJoinPull object called invoke which takes in a new Prevail object, the advection and the dimensions of the 3D array. If the sequential cutoff is

Parallel algorithm

Prevail is a class that extends Recursive task and returns a 1D array containing the total sum of all the wind vectors in advection. It takes in the 3D advection array, a lower and upper bound, which is initially zero and the total dimension, and the 3 dimensions. When the forkJoinPull is invoked, compute in Prevail is called. Within compute, the difference between the upper and lower bounds are compared to the sequential cutoff, which is set as a final variable. If the difference is less than the sequential cutoff, the sum of the vector components will be calculated the same way as was done in SeqCloudData.

If the difference between the two bounds is still larger than the sequential cutoff, then Prevail will be called recursively with the bounds being divided by half each time. This is done for the right half and left half of the array recursively dividing each side be two. The sums for each side are added up and the result is sent back to the ParallelCloudData.

The parallel Classify class uses the same concept. However, it extends RecursiveAction and no result is returned. Instead, it fills up the classification array with the correct values.

Time measurement and Speed Up

The time was captured each time either prevailingWind or classification was run in the sequential or parallel programs. The programs were each run five times. The sequential cut off for the parallel program was varied from 5 to 100000 for different data sizes (TODO: insert data variation). The same data sizes and sequential cut-offs were run on two separate machines. One a remote server and the other a laptop (see appendix for machine specification)

The speed up was calculated as the average of the 5 sequential divided by the time to do the parallel i.e $\frac{sequential\ time}{parallel\ time}$. This value was compared to the expected output using Amdahl's law $\frac{1}{1-p+p/n}$ where p is the ratio of the code that is parallel. P was calculated as $1 - \frac{sequential\ cut-off}{size\ of\ data}$.

The process of benchmarking was automated in a bash script that contained the sequential cut-offs as variables and ran the parallel program 5 times for each variable.

Results and analysis

The graphs below show how the different data sizes affected the speed Up. The graphs all show the Speed up against the percentage of sequential. They all contain the prevailing speed up, classification speed up and expected speed up based on the formula above.

From the graphs, the point where the expected speed up and either the classifying or prevailing speed up intersect is when the cut off is 10000. This corresponds to a speed up 1.

The best performance was on the server with a data set of 20x512x512 with a speed up of 1.8x. The laptop only attained a speed up of 1x in the best case. The performance on both machines increases as the data sets increase. In all the graphs, as the sequential cut-off increases, the Speed up increases. Therefore, for this algorithm, sequential code will run faster than parallel.

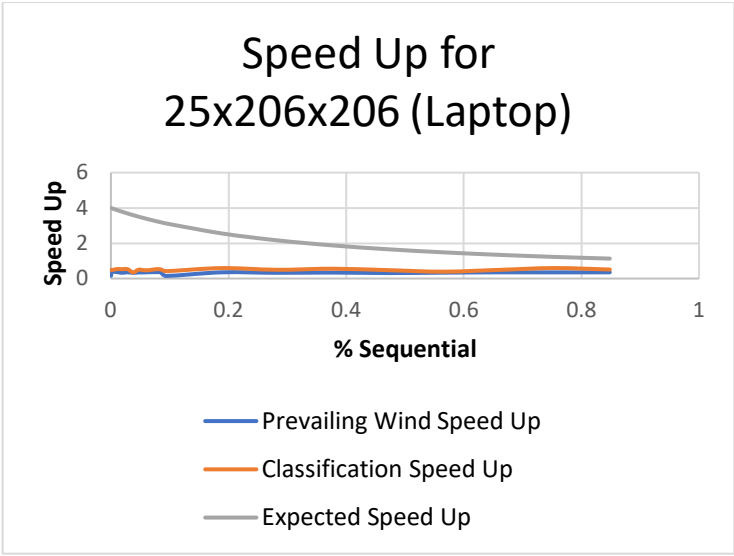

Figure 1: Speed up for 25x206x206 for laptop

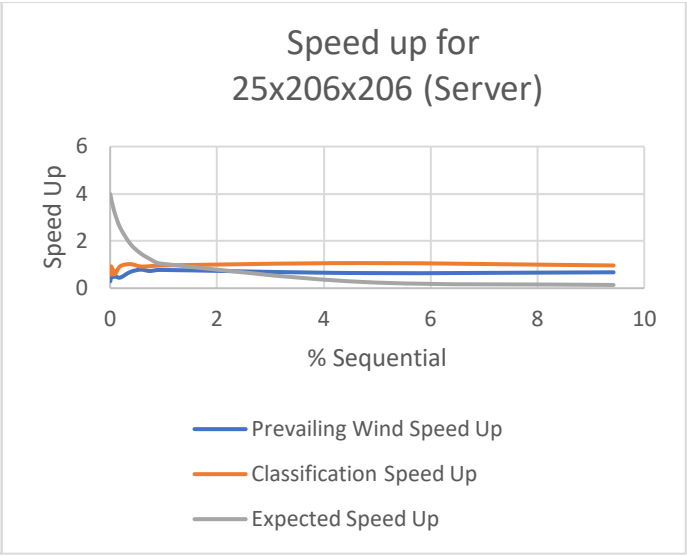

Figure 2: Speed up for 25x206x206 for Server

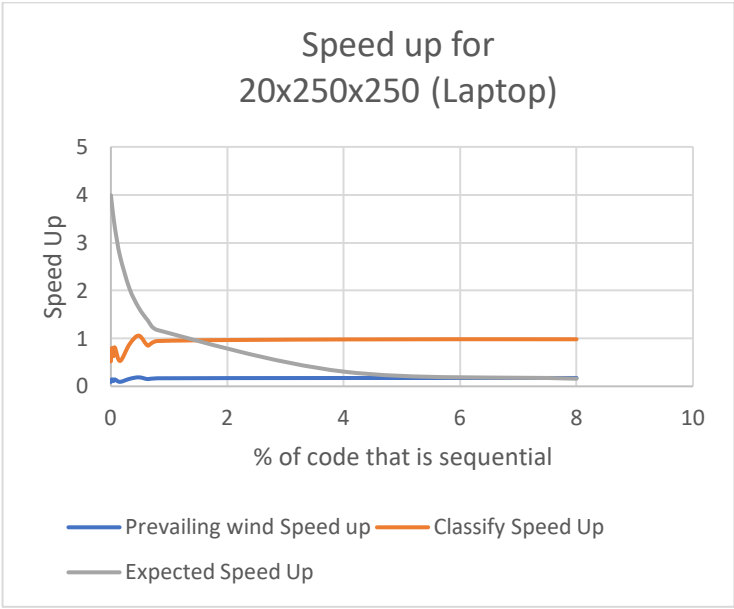

Figure 3: Speed up for 20x250x250 for laptop

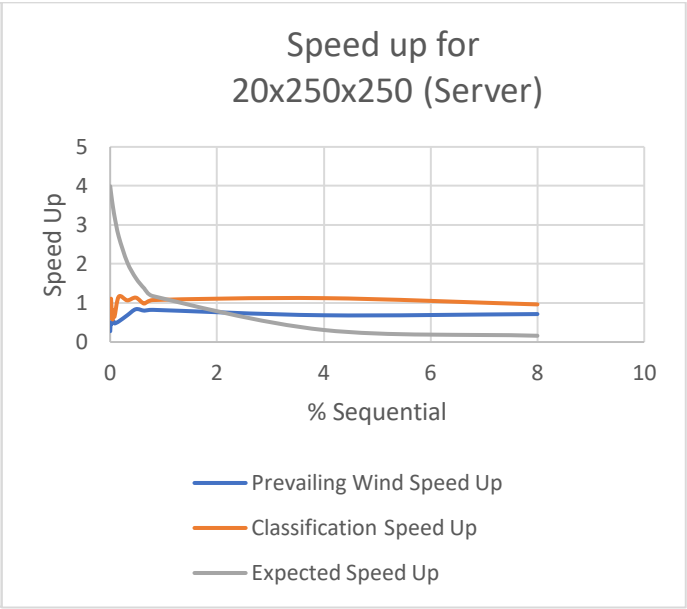

Figure 4: Speed up for 20x250x250 for Server

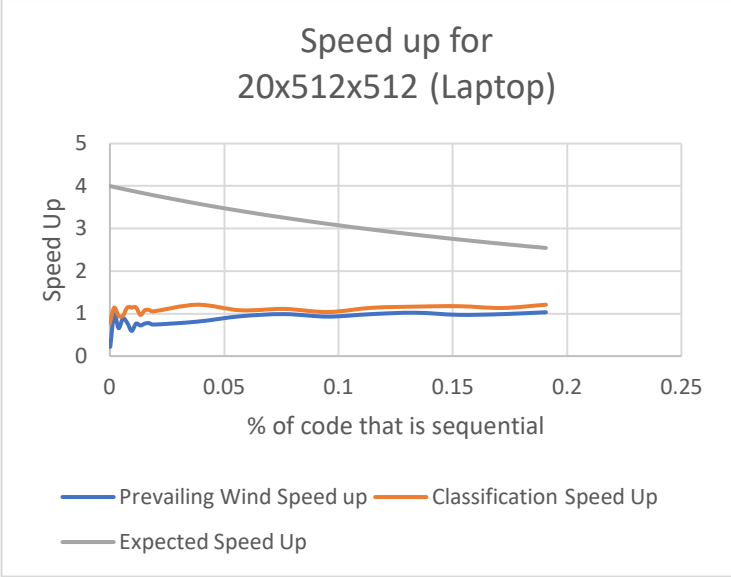

Figure 5: Speed up for 20x512x512 for laptop

Figure 6: Speed up for 20x512x512 for Server

Conclusion

At the beginning of the project, it was expected that the sequential code would run slower and the speed up would be as close to the expected. There may have been factors that influenced the speed. The laptop used was dual booted and other processes were running as the code was being executed. This may not have had such a large impact, but it may have affected the speed of the code. The conclusion is that the parallel algorithm that was used was not optimal and needs further revision. A different algorithm must be devised that ensures that the speed up is at least greater the 2X.

Appendix

Computer Architectures used for the experiment

Computer 1: laptop

Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
CPU(s): 8
On-line CPU(s) list: 0-7
Thread(s) per core: 2
Core(s) per socket: 4
Socket(s): 1
NUMA node(s): 1
Vendor ID: GenuineIntel
CPU family: 6
Model: 158
Model name: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
Stepping: 9
CPU MHz: 900.054
CPU max MHz: 3800.0000
CPU min MHz: 800.0000
BogoMIPS: 5616.00
Virtualization: VT-x
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 6144K

Computer 2: remote server (nightmare)

Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
CPU(s): 8
On-line CPU(s) list: 0-7
Thread(s) per core: 2
Core(s) per socket: 4
Socket(s): 1
NUMA node(s): 1
Vendor ID: GenuineIntel
CPU family: 6
Model: 44
Model name: Intel(R) Xeon(R) CPU E5620 @ 2.40GHz
Stepping: 2
CPU MHz: 2000.000

CPU max MHz:	2401.0000	L1d cache:	32K
CPU min MHz:	1600.0000	L1i cache:	32K
BogoMIPS:	4787.87	L2 cache:	256K
Virtualization:	VT-x	L3 cache:	12288K