Introduction

This project centres around weather simulation for the purposes of prediction. This is a complex task that often needs to be completed within a specific time limit in order to be useful.

For this project, I 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.

The input is 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. An x and y co-ordinate as well as an uplift value (represents the rate at which air shifts upwards or downwards).

Algorithm Overview

To achieve the task specified above, I will parallelise the problem in order to speed it up using Java's Fork/Join framework by calculating the cloud classification and prevailing calculations using a divide-and-conquer algorithm.

The framework first "forks", recursively breaking the task into smaller independent subtasks until they are simple enough to be executed asynchronously. After that, the "join" part begins, in which results of all subtasks are recursively joined into a single result, or in the case of a task which returns void, the program simply waits until every subtask is executed. To provide effective parallel execution, the fork/join framework uses a pool of threads called the ForkJoinPool, which manages worker threads of type ForkJoinWorkerThread.

Expected Performance

Given that testing will occur on processors with 4, 4 and 2 cores, Amdahl's law suggests we should expect a speed up of at most 4,4 and 2 respectively. Given that overheads are present, it is extremely unlikely that it will be 8 and 4, but it will not be more than that.

Experiment

Methods

Classification Problem

The most challenging part of the assignment was the cloud classification task. I created the setClassification method to populate the classification Array. The main aim of this method was to calculate the local average as is shown below

1. The Local Average calculation

Given the (i, j, t)th position in the array of size (x,y,z), note that 0<i<x, and 0<j<y.

sum, neighbours = 0

for (integer a = max(0, i-1), i<min(x,i+2),i++){

for(integer b = max(0, j-1), i<min(y,j+2),j++){

sum = sum + vector[a,b,t]

Neighbours++

Average = sum/neighbours

Although there is a nested for loop, it can only run for a maximum of 3 times on each loop, or 9 total iterations. The t parameter remains constant as the local average is only calculated across one time stamp, hence the need for only two for loops. Min and Max functions enable the algorithm to account for the boundary cases like if the central element is in the corners or on the edges.

Once the local average has been calculated, The magnitude of the local average is then calculated and compared to uplift values in an if/if else/else block of statements to determine the element's classification.

Parallelisation

Given a min position and a max position of a 1 dimension array (that is equivalent to the 3 dimensional array)

If (max-min < SEQUENTIAL_CUTOFF)

The local average is calculated and classified

For each element in the 1d array from min to max, the vectors are summed

Else

two new parallel threads are created from min to min+max/2 called left

and (min+max)/2 to max called right.

Left is forked, right is computed and when it is completed, left joins.

(This is the recursive divide and conquer algorithm that will speed up the program.)

Timing

I used the tick() and tock () to time. I placed the tick method directly after the read method and the tock method before the write method to get an accurate reading of processing time.

Approach

My approach was to run the sequential and parallelised programs on incrementally larger dataset sizes between 450 thousand and 45 million, with multiple sequential cutoffs tested for the parallelised program. This was repeated for all three architectures tested.

Validation and Testing

The algorithm was validated by running the programs with numerous small inputs with easily calculated averages done by hand. It also got the correct solution for the largesample_input.txt file. By using the fork/join framework, no race conditions occurred and the general thread was designed to refer only to its own variables, and not shared variables which meant there were no static totals to update. The parallelization, as shown later, also provided speed up hence being efficient. To improve the accuracy of the timing, I had the first 11 tries discarded to ensure the data was in

Cache. After this step I calculated the average time over 100 attempts to make the time as accurate and variance free as possible.

The project was tested on three architectures, which shall be called Architecture 1,

Architecture 2 and Architecture 3 in the rest of the report.

Architecture 1:

Intel® Core™ i3-8100 CPU @ 3.60GHz × 4 (4 Logical cores)

Architecture 2:

Intel® Core™ i5-7400 CPU @ 3.00GHz × 4 (4 logical cores)

Architecture 3:

Intel® Core™ i7-5600U CPU @ 2.60GHz × 2 (4 logical cores)

Results

Figure 1 Architecture 1

No. of elements	**Sequential Time (ms)**	**Average Time in ms (with Sequential Cutoff)**					
450K	26	5 (1K)	5 (2K)	5(5K)	5(10K)	5(50K)	5(100K)
1.25M	60	14(2.5K)	14(5K)	14(12K)	14(24K)	14(120K)	14(240K)
5M	240	56(5K)	56(10K)	57(25K)	58(50K)	57(250K)	58(500K)
10M	850	230(10K)	233 (20K)	232(50K)	232(100K)	235(500K)	234(1M)
45M	1875	509(50K)	543(100K)	542(250K)	541(500K)	542(2.5M)	544(5M)

Figure 1 above shows the average times in ms at increasing sequential cutoffs for increasing data sizes for Architecture 1.

Figure 2 Architecture 2

No. of elements	**Sequential Time (ms)**	**Average Time in ms (with Sequential Cutoff)**					
450K	25	6 (1K)	6 (2K)	5(5K)	6(10K)	6(50K)	5(100K)
1.25M	70	16(2.5K)	16 (5K)	16 (12K)	16 (24K)	16 (120K)	16(240K)
5M	250	65(5K)	65 (10K)	66(25K)	66(50K)	67 (250K)	67(500K)
10M	918	265(10K)	266(20K)	270(50K)	268(100K)	269(500K)	270(1M)
45M	2040	619(50K)	630(100K)	625(250K)	621(500K)	619(2.5M)	623(5M)

Figure 2 above shows the average times in ms at increasing sequential cutoffs for increasing data sizes for Architecture 1

Figure 3 Architecture 3

No. of elements	Sequential Time (ms)	Average Time in ms (with Sequential Cutoff)					
450K	0	0(1K)	0(2K)	0(5K)	0(10K)	0(50K)	0(100K)
1.25M	89	15(2.5K)	15(5K)	15(12K)	15(24K)	15(120K)	15(240K)
5M	230	92(5K)	93(10K)	97(25K)	91(50K)	99(250K)	101(500K)
10M	896	402(10K)	392(20K)	391(50K)	401(100K)	400(500K)	395(1M)
45M	1907	811(50K)	830(100K)	822(250K)	820(500K)	823(2.5M)	825(5M)

Figure 3 above shows the average times in ms at increasing sequential cutoffs for increasing data sizes for Architecture 1

Figure 4

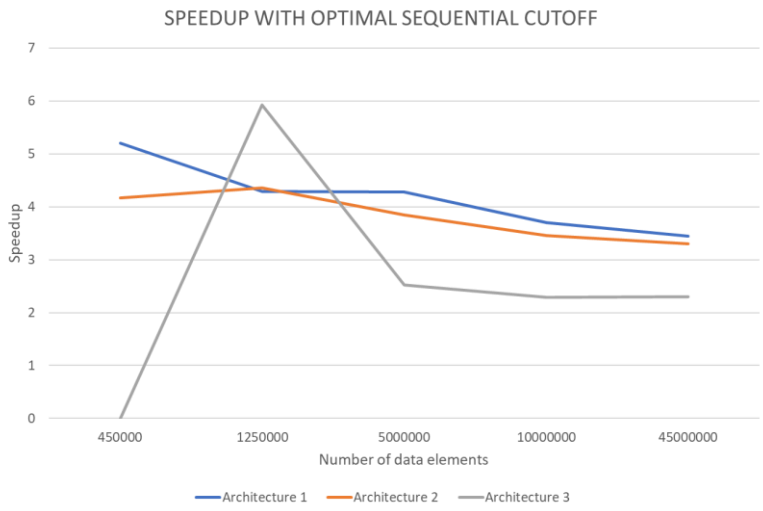

Figure 4 above shows the speedup between architectures at their optimal sequential cutoff.

Figure 5

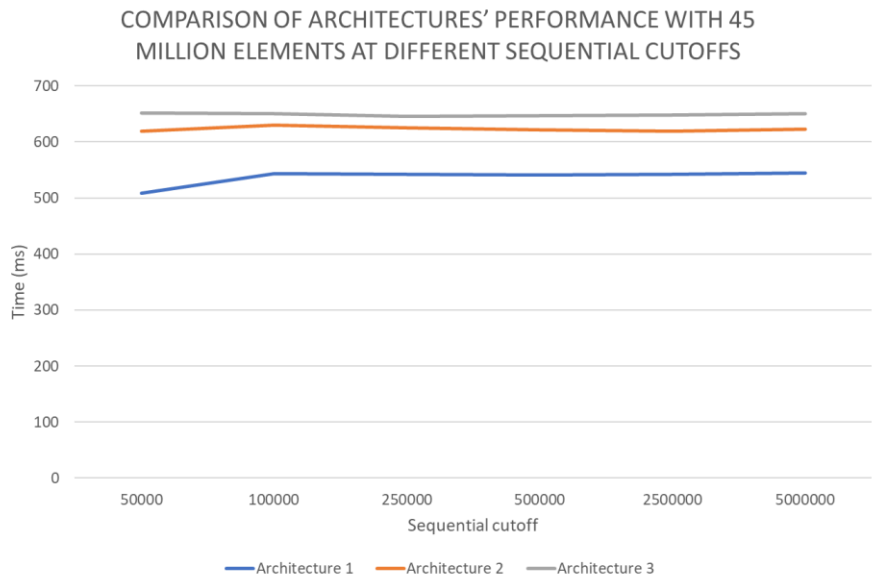

Figure 5 above shows a comparison of the architectures' performances with 45 million elements at different sequential cutoffs.

Discussion

Looking at figure 4, all three architecture display a speedup of greater than X2 at every level. This mean computation is being significantly sped up and it is certainly worth using parallelisation for this problem. Figure 4 also shows that the parallel program performs well over all the datasets I tested and the maximum speedup attained is just under 6 with Architecture 3. This is greater than the ideal time for a dual core but given the smaller data size, there may have been system interference at play. It also worth noting that Architecture 3 as hyperthreading and this may be a factor.

I estimate the optimal sequential cutoff for a data size X is between X/1000 to x/10.

The optimal number of threads for architecture 1 is ,Architecture 2 is and Architecture 3 is

Conclusions

For problems that have answers that are not order-dependent during calculation, parallelization by divide and conquer is a well-suited approach to take. Parallelisation gives speed up on all data sizes, provided the sequential cutoff is chosen carefully.

Although speedup is experienced by all data sizes, larger data sizes gain more from parallelization as they do not suffer as much from the overhead costs of the many threads that smaller data sizes do The sequential cutoff must be less than the data size.