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

Oyama C. Plati†
CSC2002S 2019
University of Cape Town
South Africa
†PLTOYA001

***Abstract*—In this project we designed programs to accomplish the task of cloud classification. The first part of the project was designing a sequential solution. The second part was designing a parallel solution that provided optimisation through correctness and speedup. The parallel program was implemented using the Java Fork/Join framework.**

I. Introduction

The aim of the project was to implement cloud classification which is a complex task that needs to be completed fast in order to be useful. Hence it is problem well suited for parallel programming [1]. The parallel program used divide and conquer algorithm for three dimensions by splitting the work amongst threads to execute smaller parts of the whole and once done rejoining the parts to make the final solution. This was done with the Java Fork/join framework.

The aim of this project was to implement cloud classification using the Java Fork/Join framework and achieve correctness and speedup when compared to a sequential implementation.

The parallel program was compared to be compared to a sequential program used to solve the same problem. A series of optimisations were expected with the parallel programming through correctness and speedup. The comparisons were done for floating point precision, ranges of data sets, number of threads and different types of architectures. For each category were to measure the timing of prevailing wind and cloud classification calculations done in both solutions.

For a two core machine we expected a speed up of two times when compared with the sequential program. For a four core machine we expected 4 times speed up when compared with the sequential program.

II. Methodology

A. Hardware

The following lists the hardware used in this project

- Vim
- Java
- make
- Multi-core machine

B. Implementation

We implemented a sequential solution to cloud classification in *CloudDataAnalysis.java*. The sequential implementation used loops to traverse through the three dimensional data and do calculations.

Next we implemented the parallel solution to the program by using the Java Fork/Join framework to parallelise the cloud classification and prevailing wind calculations using a divide-and-conquer algorithm. The following code snippets show the classes that were created *PrevailingWind.java*, *CloudClassification.java* and *CloudDataParallelAnalysis.java*.

C. Experiment Procedure

The performance of the implementation was measured by timing the prevailing wind and cloud classification calculation code. The timing was measured at least five times for the sequential implementation and parallel implementation. The result were recorded.

The speed up was calculated using the following equation

$$Speed - up = \frac{SequentialTime}{ParallelTime} \tag{1}$$

where *Sequential Time* referred to time taken by the sequential solution to complete for the prevailing wind and cloud classification code combined. The *Parallel Time* referred to time taken by the parallel solution to complete for the prevailing wind and cloud classification code combined.

The accuracy was calculated using the files by comparing the outputs in the file created when both implementation are executed using the following class *Accuracy.java*.

```
import java.io.File;
import java.io.IOException;
import java.util.*;
import java.io.FileWriter;
import java.io.PrintWriter;

class Accuracy{
    public static void main(String []args){
        try{
            Scanner file_original = new Scanner(new File(args[0]), "UTF-8");
            Scanner file_generated = new Scanner(new File(args[1]), "UTF-8");

            int i =0;
            file_original.nextLine();
            file_original.nextLine();
            file_generated.nextLine();
            file_generated.nextLine();

            while (file_original.hasNext()){
                if (file_original.nextInt() != file_generated.nextInt()){
                    System.out.println(i);
                }
                i++;
            }
        }
        catch(Exception e){e.printStackTrace();}
    }
}
```

These measurements were done for each project benchmark such as the varying data set sizes, number of threads and different machine architectures. The benchmarks were tested on a 2 core and 4 core machine. The data set sizes (times steps - x dimension - y dimension) were extremely small: $(5 - 100 - 100)$, $(20 - 512 - 512)$, $(25 - 600 - 600)$, $(25 - 1000 - 1000)$ and $(25 - 1500 - 1500)$. We generated the data set for testing using *generateDataset.java* to a file windGenerated.txt. The effect of the sequential cut off was tested by setting the data size constant $(30 - 600 - 600)$ and we measured the performance of the sequential solution and then measured the performance of the parallel solution for different sequential cut of sizes from the size of the data set to a point were the speed up of the program did not show anymore improvement. The results were recorded for time and speed up of the parallel implementation.

III. Results and Discussion

A. Two core machine

In this stage of the project we tested the performance of the sequential and parallel solution on a two core machine.

1) Data size: The following shows the results from measuring the performance of the sequential and parallel solution when we tested the data set benchmark on a two core machine.

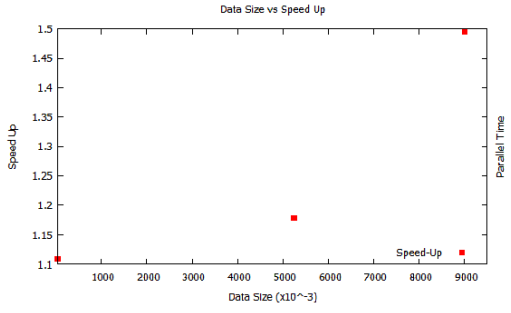

Fig. 1: Speed up graph for 2 core machine for varying data set size. (Red squares in plot).

The times were of the prevailing wind and cloud classification time added together and averaged for ten measurements for each data set size when testing the sequential and parallel solution. As seen in table I, on the two core machine the data set size do not exceed 9 million points. The compiler would throw a run time error for GC Overhead Limit Exceeded Error which was throw because the JVM spent too much time on garbage collection and freed up very little heap space. From the data we had the parallel implementation showed an increase in speed up as calculated using equation (1).

TABLE I
Sequential and Parallel solution performance for varying Data size

Dataset size	Sequential Time	Parallel Time	Speed Up
50 000	0.0335	0.0320	1.109
5 242 880	0.4061	0.3447	1.178
9 000 000	0.6152	0.4113	1.496

2) Sequential Cut-off: The following shows the results from measuring the performance of the sequential and parallel solution when we tested the sequential cut-off or increasing the number of threads benchmark on a two core machine.

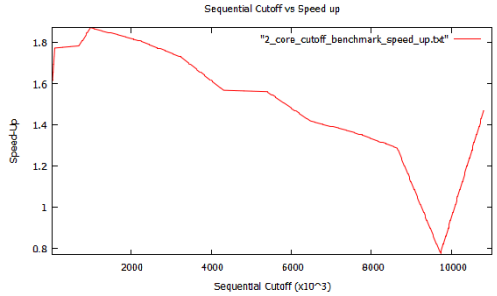

Fig. 2: Speed up graph for 2 core machine for varying sequential cut off. (Red line in plot).

The sequential cut-off benchmark was tested for a fixed data set size of $(30 - 600 - 600)$ or 10 800 000. We first tested for the case were the sequential cut-off was 10 800 000. Therefore reduced the parallel case to sequential execution and we recorded the results. From then on we reduced the sequential cut-off i.e increasing the number of threads. As seen in table II and figure 2 the speed up decreased as fewer and fewer threads are used. Until sequential cut-off 720 000 were we stopped seeing significant speed up improvement.

3) Discussion: Looking at the results of the performance of the parallel solution we saw that on a two core machine the data set sizes for which the parallel program performed well was (9000000) in table I. The maximum speed up from the parallel program occurred also at (9000000) in figure 1 at 1.496 which is less than the expected speed up of 2 time the sequential program. If the data set size kept increasing we would have seen the expected results from the experimental measurements.

For the two core machine the optimal sequential cut-off for this problem was 1 000 000 where we achieved the highest

TABLE II
SEQUENTIAL AND PARALLEL SOLUTION PERFORMANCE FOR VARYING NUMBER OF THREADS

Sequential Cut-off	Sequential Time	Parallel Time	Speed Up
50 000	0.926	0.574	1.613
100 000	0.926	0.522	1.774
720 000	0.926	0.519	1.784
1 000 000	0.926	0.495	1.871
2 610 000	0.926	0.511	1.812
3 240 000	0.926	0.535	1.731
4 320 000	0.926	0.591	1.567
5 400 000	0.926	0.593	1.561
6 480 000	0.926	0.653	1.418
7 560 000	0.926	0.679	1.364
8 640 000	0.926	0.719	1.288
9 720 000	0.926	1.195	0.775
10 800 000	0.926	0.630	1.469

speed-up, calculating to approximately $\approx$ 11 threads as seen in table II.

B. Four core machine

In this stage of the project we tested the performance of the sequential and parallel solution on a four core machine.

1) Data size: The following shows the results from measuring the performance of the sequential and parallel solution when we tested the data set benchmark on a four core machine.

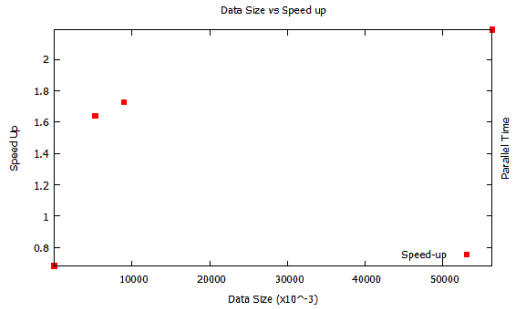

Fig. 3: Speed up graph for 4 core machine for varying data set size. (Red squares in plot).

The times were of the prevailing wind and cloud classification time added together and averaged for ten measurements for each data set size when testing the sequential and parallel solution similarly to the case of two case machine.

2) Sequential Cut-off: The following shows the results from measuring the performance of the sequential and parallel solution when we tested the sequential cut-off or increasing the number of threads benchmark on a four core machine.

TABLE III
SEQUENTIAL AND PARALLEL SOLUTION PERFORMANCE FOR VARYING DATA SIZE

Data set size	Sequential Time	Parallel Time	Speed Up
50 000	0.169	0.0247	0.684
5 242 880	0.349	0.2136	1.637
9 000 000	0.408	0.2356	1.730
25 000 000	1.0305	0.4561	2.259
56 250 000	2.2374	1.0229	2.187

TABLE IV
SEQUENTIAL AND PARALLEL SOLUTION PERFORMANCE FOR VARYING NUMBER OF THREADS

Sequential Cut-off	Sequential Time	Parallel Time	Speed Up
5 000	0.4848	0.2966	1.635
10 000	0.4848	0.3078	1.575
20 000	0.4848	0.3072	1.578
50 000	0.4848	0.2624	1.848
100 000	0.4848	0.2702	1.794
720 000	0.4848	0.2882	1.682
1 000 000	0.4848	0.2786	1.740
2 610 000	0.4848	0.2724	1.779
3 240 000	0.4848	0.3062	1.583
4 320 000	0.4848	0.3062	1.583
5 400 000	0.4848	0.3112	1.558
6 480 000	0.4848	0.2932	1.653
7 560 000	0.4848	0.3102	1.563
8 640 000	0.4848	0.3252	1.491
9 720 000	0.4848	0.3220	1.506
10 800 000	0.4848	0.3530	1.373

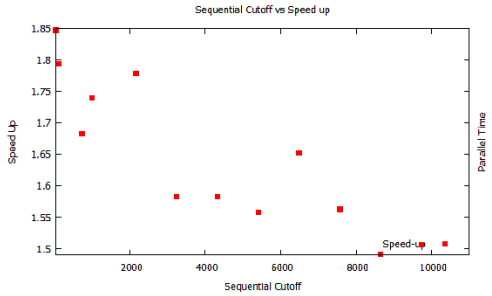

Fig. 4: Speed up graph for 4 core machine for varying sequential cutoff. (Red squares in plot).

As seen in figure 4 the smaller the sequential cut-off the higher the speed up. We saw a gradual increase in speed up until $\approx$50 000 where decreasing the sequential cut-off to lower values did not show any significant increase in speed up as seen in table IV.

3) Discussion: We tested the machine on two different machines. On the four core machine we saw better performance. The data set sizes where the parallel program performed best were sizes $\geq$ 9 million data points as seen in table III. There larger the data set the greater the speed up. The maximum speed obtained was 2.259 for a data set size of 25 million data points (see table III). The 2.259 times speed up is less than the 4 times expected speed up. The optimal number

of threads in the four core machine for 50 000 sequential cut off is 216 threads see IV.

The parallel implementations in both architectures used in the project produced speed ups of approximately twice the sequential implementations. At low levels very little work is being done by both the parallel and sequential programs, hence we were not expecting high speed up. For size of the amount of data processed for the two and four core machine we also did not expect much speed because the size of the data is relatively small. Even though parallelizing the implementation of cloud classification is possible, in this case, the dividends are simply not high enough to make significant difference in practical uses.

IV. Conclusion

We tested the benchmarks on a two core and four core machine. The two core machine had maximum speed up of 1.496. The two core machine had best speed up 1.871 when using 11 threads. The four core machine had maximum speed up of 2.259. The four core machine had the best speed up of 1.848 when using 216 threads. From the results we concluded that for our benchmarks to produce the best performance one should use the four core and higher core machine.

References

[1] S. Berman, "Parallel Programming with theJava Fork/Join framework: Cloud Classification," , Aug. 2019, version 1.