CSC2002S
Assignment 3: Parallel Programming using the Fork/Join Framework
DPLJER001
September 2018

Introduction

This report details the problem, methods and results of Assignment 3: Parallel Programming with the Fork-Join Framework. The structure is as follows; first, an introduction to the problem and a description of the parallel algorithm used in the solution and its expected improvement to performance, followed by the approach and methods used to solve the problem, ending with an analysis of the results generated from experiments run on different architectures.

The aim of the project is essentially, given input data in *<input_file_name>.text* format, to calculate the average daily sunlight exposure of 1'000'000 tree canopies, as well as a total for each individual tree, in a 3000x3000 meter grid representing a natural terrain (somewhere in California). The specific objective with regards to the parallelisation of the problem is to break the problem of summing over 1'000'000 objects up into a set of smaller problems using a divide-and-conquer algorithm; creating multiple threads recursively to tackle each small piece of work concurrently in order to increase performance (speedup) relative to a regular sequential algorithm running in a single thread. As per instruction the Fork/Join framework was deployed in the solution to the problem. The framework is designed for work which can be broken up into smaller pieces and solved recursively, allowing one to take advantage of multiple processors, using all available processing power. The framework extends an *ExecutorService* which essentially distributes tasks to threads in a "thread pool", allowing threads to execute pieces of work concurrently on different processors, and even "steal" work from other threads if they run out of things to do. **Fig 1.** illustrates how the divide and conquer, recursive style algorithm works; given an array of things to be summed, the array is halved recursively, giving each new half to a new thread of work until the interval size is below a sequential cutoff at which point each thread performs a sequential sum over its own sub-interval of the array, returning the result. The results are fed back up the recursive 'tree' and summed together to

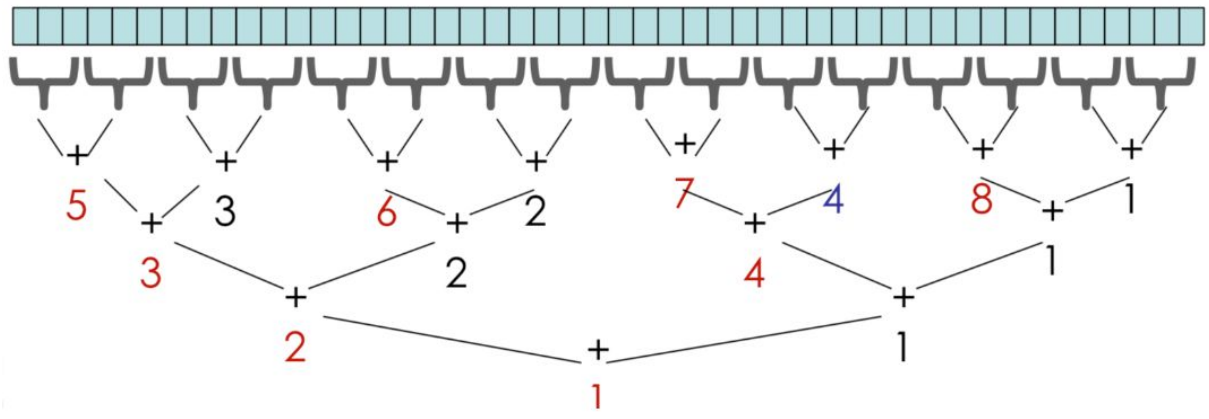


Fig 1. A visual representation of a divide and conquer algorithm using multiple threads. At each step a new thread is created, given half the work and merged off until sequential cutoff is reached.

get the total for the entire array. In the solution to this specific problem, a default sequential cutoff of 500 is used, and given the size of items to be summed is 1'000'000 we can calculate the proportion of work, P, which is concurrent as

$$P = 1 - \frac{\textit{Number of items to be added sequentially}}{\textit{Total number of items}} = 1 - \frac{500}{1000000} = 1 - 5 \times 10^{-4} = 0.9995$$

Using this we can calculate the expected speedup of the parallel summation algorithm relative to a sequential algorithm using Amdahl's Equation. If we run the program on a machine architecture which has N processors (cores), the speedup of a parallel program is given by the reciprocal of the the sequential portion of the algorithm, $1-P$, added to the parallel portion divided by the number of cores, $\frac{P}{N}$:

$$\textit{Speedup} = \frac{1}{1-P+\frac{P}{N}}$$

Then, for two and four processors respectively, we can expect an approximate, relative speedup in performance of

$$\textit{Speedup}_{(2\ cores)} = \frac{1}{1-0.9995+\frac{0.9995}{2}} = 1.9990005 \approx 2 \times \textit{speedup}$$

$$\textit{Speedup}_{(4\ cores)} = \frac{1}{1-0.9995+\frac{0.9995}{4}} = 3.994008... \approx 4 \times \textit{speedup}$$

Fig 2. Confirms out expectation and illustrates the types of increase in performance we may expect from a parallel algorithm using a given number of processors.

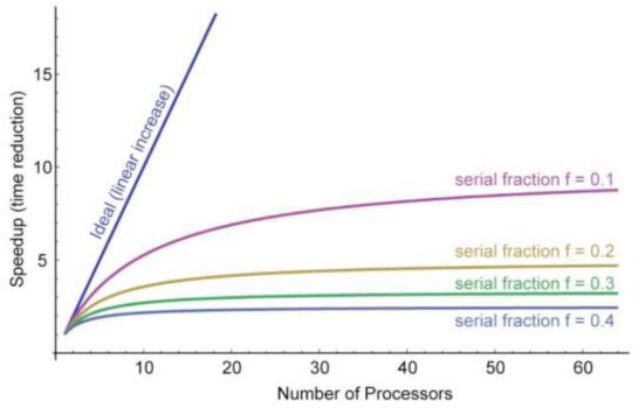


Fig 2. A visual representation of Amdahl's law given the Sequential (serial) proportion of an algorithm, and a number of processors.

Two important caveats to the expected speedup calculated above using Amdahl's equation are:

1. Speedup will vary depending on the problem size (the size of the array to be summed in this case). In fact, we should expect that speedup will be correlated positively with the problem size, which is the case in the results of the experiments detailed below.

2. Speedup will be hampered by overhead generated by threads (work done by a processor to create and destroy a thread in memory) and other peripheral calculations done by threads which are not considered in theoretical calculations like the ones above.

Therefore, those are the ideal speedup ratios we can hope for in this particular scenario.

Methods

The following section provides a description of the approach used to solve the problem with details on the parallelisation, followed a description of the method of validation of correctness, timing, measure of relative speedup of the program, and finally, details about the experiments run and machine architectures used.

Solution Approach

In order to ensure the program ran with maximum efficiency, the approach to the solution was to ensure all expensive operations (specifically String manipulation) were done prior to feeding the data into the parallel algorithm. To this end, as seen in **Fig. 3**, the data for the per-square-meter sunlight exposure of the terrain was read from file and stored in a 3000x3000 sublight matrix of floating point values, and the data pertaining to the 1'000'000 individual trees, x,y coordinates and the dimension of the canopy, were stored in *tree* objects, which in turn were stored in a *tree* object array. The *tree*

```
public static bundle getTreeMatrix(String filename) throws IOException{

    File file = new File(filename); // load input text file
    BufferedReader br = new BufferedReader(new FileReader(file));

    int size = Integer.parseInt(br.readLine().split( regex: " ")[0]); // dimension of square sunlight matrix
    StringTokenizer digits = new StringTokenizer(br.readLine());    // array of sunlight exposure per square meter

    float[][] sunMatrix = new float[size][size];

    for(int row=0; row<size ; row ++){
        for(int col=0; col<size; col ++){
            sunMatrix[row][col] = Float.parseFloat(digits.nextToken()); // fill sunlight matrix
        }
    }

    int numTrees = Integer.parseInt(br.readLine()); // total number of trees
    tree[] treeList = new tree[numTrees];

    for(int k=0; k<numTrees ; k++){
        String[] temp = br.readLine().trim().split( regex: " ");    // x-coordinate, y-coordinate, dimension of canopy
        treeList[k] = new tree(Integer.parseInt(temp[0]),          // create new tree object
                               Integer.parseInt(temp[1]),
                               Integer.parseInt(temp[2]));
    }
    return (new bundle(treeList, sunMatrix)); // return sunlight matrix and array of tree objects
}
```

Fig 3. The getTreeMatrix() method used to read the data from file.

array and sunlight matrix were in turn stored in a *bundle* object which was then passed into the parallel program sumArray in the Fork/Join pool (more on this below).

The reason arrays and matrices work well for this kind of problem is due to their pass-by-reference quality; the variables pointing to the sun matrix and the *tree* array in memory can be passed down the recursive 'tree' of threads, allowing all threads to read from the shared objects in memory without having to use lots of working memory by making multiple copies of the same thing. Also, since threads are only reading and not writing from the shared data (the sun matrix), there is no danger of race conditions occurring. In terms of required output, to ensure the total sunlight value for each tree can be written to file in the same order they are read in after the parallel/recursive portion of the program runs, each *tree* object saves the total for that particular tree as an instance variable, allowing, by virtue of pass by reference, the total per tree to be accessed afterwards in the main thread.

After retrieving all the data from file and storing both the sunlight matrix and the array of *tree* objects in the *bundle* object, the main method in the driver class, *treeSumParallel*, calls the *sum()* method[1], passing it the *bundle* object, which creates a new Fork/Join pool in order to begin distributing tasks

```
/*
* sum creates new fork-join pool for threads
* calls invoke() on SumArray object which inherits
* from RecursiveTask<V>
*/
static float sum(bundle b){
    return fjPool.commonPool().invoke(new SumArray( lo: 0, b.getTrees().length, b));
}
```

```
@Override
protected Float compute(){

    if ((hi - lo) < Constants.SEQUENTIAL_CUTOFF) {  // minimum tasks a thread must handle
        return treeUtils.sequentialSum(b, lo, hi);
    }

    else{
        SumArray left = new SumArray(lo, hi: (hi+lo)/2, b); // new SumArray (RecursiveTask) object
        SumArray right= new SumArray( lo: (hi+lo)/2, hi, b);

        left.fork(); // branch off thread
        float rightAns = right.compute(); // main thread does this
        float leftAns = left.join(); // wait until left is done

        return leftAns + rightAns ; // merge left and right threads
    }

}
```

Fig 4. The sum() method which creates a new Fork/Join pool, passing in a new SumArray object which contains the compute() method responsible for recursively creating new threads of work (as per Fig 1.)

[1] For the purposes of this assignment, the main method in the driver class actually runs **eight iterations** of the parallel sum program, allowing the **first three for optimization** and then taking **high precision timing for the remaining** five iterations to report the average time taken per iteration.

to worker threads. The Fork/Join pools *invoke()* method is invoked to initialise the pool, and takes as its argument a new *SumArray()* object, which inherits from *RecursiveTask<V>*. These objects can be thought of as threads which are created recursively in the sense that a *SumArray()* objects *compute()* method which is called by the Fork/Join pool object creates new SumArray() objects in turn, essentially splitting the work in two (see the *left.fork()* instruction in the *compute()* method **Fig 4.**). Each new *SumArray() object* is passed a *bundle* object containing variables which reference the *tree* array and sunlight matrix in memory, as well as the variables *lo* and *hi* which give essentially tell the *SumArray()* object which sub-interval of the *tree* array to sum. Eventually the halving of the intervals passed recursively to the new SumArray() objects reaches below a certain point (*hi-lo < Constants.SEQUENTIAL_CUTOFF*) at which point the *sequentialSum()* method in the *treeUtils()* class is invoked, and the thread will then sum over an interval of the *tree* array which is the size of the SEQUENTIAL_CUTOFF variable. **Fig 5.** Shows the *sequentialSum()* method in which the total for each individual tree is calculated and stored in a *tree* object, and how the total for the specific sub-interval of the *tree* array is calculated and returned; passing it back up the recursive 'tree' to be added

```
public static float sequentialSum(bundle b, int start, int stop){

    tree[] treeList = b.getTrees();          // array of tree objects
    float[][] sunMatrix = b.getSunMatrix(); // sunlight matrix (p/square meter)

    double totalSun = 0.0;

    for(int i=start; i<stop; i++){
        float sum = 0;

        tree tempTree  = treeList[i];
        int x = tempTree.getX(); // coordinates of top left corner of tree in matrix
        int y = tempTree.getY();
        int dim = tempTree.getDimension(); // dimension of tree canopy

        int Xdim = Math.min(x+dim, sunMatrix.length); // to handle "overflow" in the X-axis
        int Ydim = Math.min(y+dim, sunMatrix.length); // to handle "overflow" in the Y-axis

        for(int row=y; row<Ydim; row ++){
            for(int col=x; col<Xdim; col ++){
                float sunLight = sunMatrix[row][col]; // get sun exposure value at point (x,y)
                sum += sunLight;
            }
        }
        tempTree.setSun(sum); // set total sunlight exposure for a tree
        totalSun += sum; // add sunlight exposure for tree to total sum
    }
    return (float)totalSun; // return total sum
}
```

Fig 5. The *sequentialSum()* method used by a thread to calculate the total sum for a portion of the tree object array in sequence and return the result.

together to all the other smaller sums.

A minor challenge encountered with large the large sum was the loss of accuracy when using floating point type variables to store large values, so all values are summed up as double variables, and type cast to float upon return. After the recursive, parallel sum is finished the grand total of sunlight received by all trees on the terrain is returned to the *sum()* method called in the driver

class, and the average is acquired by dividing the total by the length of the *tree* array. The total for each tree recorded in each *tree* object in the *tree* array which can be accessed from the driver class. All results are written to file using the *toFile()* method in the *treeUtils* class.

As stated above, the parallel portion of the program maintains its efficiency by only performing operations using numerical values read from the matrix and the *tree* object array, ensuring that each invocation of *sequentialSum()* creates the minimum possible number of new variables in memory. The only efficiency challenge experience during coding was in an initial attempt to read from the text file using *Scanner* which was incredibly slow, so *BufferedReader* was used instead, proving to be very efficient.

In order to re-run the solution program *treeSumParallel*, read the README folder in the root directory for this project.

Validation of results

As a test of validation, the *sample_output.txt* file from Vula and the output .txt file generated by the parallel program, given the sample_input.txt file as input, were extracted line by line in a loop. Every line in both files a numerical value, so the digits were parsed as floats and subtracted from one another, with any non-zero result being reported. As far as the this test goes, the program proved correct in it's output. The Test.java source code is in the root directory and may be used to validate the results.

Timing

In order to time the parallel (and sequential) algorithm the *System.currentTimeMillis()* method was invoked to record the time before and after the invocation of the *sum()* method in the driver class, and the difference in time recorded as the speed of a single run.

```
// run parallel sum 8 times, take the average time of last 3 runs
System.out.println("Running 8 iterations of parallel sum (allowing first 3 for optimisation)");
for(int i=0; i<8; i++){
    System.gc();        // call garbage collector before parallel sum algorithm to maximise efficiency
    tick();             // record start time
    total = sum(b);     // calculate sum in parallel
    if(i>2)
        time += tock(); // time taken for one run
}
```

```
public static void tick(){
    startTime = System.currentTimeMillis();
}

public static float tock(){
    return (System.currentTimeMillis() - startTime) / 1000.0f ;
}
```

Fig 6. Using the System.currentTimeMillis() method to time the speed of the Parallel algorithm.

Measure of Speedup

The speedup in performance of the parallel algorithm relative to the sequential algorithm can be measured by taking the ratio between the average time taken over 5 sequential runs (the time taken for one processor) divided by the average time taken over 5 parallel runs (the time taken for N processors). As an example, here are two outputs from the treeSumParallel solution program on a dual core architecture; one in parallel (a sequential cut off of 500) and one sequential (a sequential cutoff of 1'000'001):

```
Jeremys-MacBook-Pro:DPLJER001(A3) jeremyduplessis$ make run_main
java -cp bin treeSumParallel "data/main_input.txt" "data/main_out.txt"
Running treeSumParallel...
Reading data from data/main_input.txt
Sequential cut off: 1000001
Running 8 iterations of parallel sum (allowing first 3 for optimisation)
Writing results to data/main_out.txt
Done.
Average Time Taken: 0.038 Seconds
```

```
Jeremys-MacBook-Pro:DPLJER001(A3) jeremyduplessis$ make run_main
java -cp bin treeSumParallel "data/main_input.txt" "data/main_out.txt"
Running treeSumParallel...
Reading data from data/main_input.txt
Sequential cut off: 500
Running 8 iterations of parallel sum (allowing first 3 for optimisation)
Writing results to data/main_out.txt
Done.
Average Time Taken: 0.0194 Seconds
```

The speedup in this case is given by:

$$Speedup_{(2\ cores)} = \frac{Sequential\ Time}{Parallel\ Time\ (Dual\ Core)} = \frac{0.038\ seconds}{0.0194\ seconds} = 1.95876... \approx 2 \times increase\ in\ performance$$

Which is what we would typically expect given our previous calculations.

Experiments

A *treeExperiments* class was created to automate the running of experiments to deduce the speedup in performance of the parallel algorithm on various architectures. The method of experimentation is as follows:

- Create a subset of the data in the *sample_input.txt* file on which to run the experiment. Subsets of size 250'000, 500'000, 750'000 and 1'000'000 were used.
- Test the speed of the program when run sequentially. This is achieved by calling the *sequentialSum()* method directly in the driver class and summing over the entire data set (see the *treeExperiments* class).
- Test the speed of the parallel algorithm for various values of the SEQUENTIAL_CUTOFF parameter(from 50 to 1000 in steps of 25); run the program in parallel for each given value of the SEQEUNTIAL_CUTOFF, taking an average running time over 17 runs per value.

The experiment can be rerun on any architecture by entering the command

$make run_experiments

from the root directory. Results are saved in a *.txt* file in the */testResults* folder. The experiments were were run on two different architectures:

1. Macbook Pro 2017, 2,3 GHz i5 processors (3,1 GHz at a boost), Dual Core
2. Linux Desktop Machine (second year labs at UCT), 3,4 GHz i5 processors, Quad Core

Below is an analysis of the results of the experiments run on these architectures, and an interpretation of the results produced.

Results & Discussion

The images below represent the results of the experiments run on the two architectures. Figures 1-8 illustrate the results of the experiment for a particular data set size(250'000 to 1'000'000); the plots show the average time taken in seconds for the sequential algorithm versus the average time taken in seconds for the parallel algorithm over a range of sequential cut offs, from 50 to 1000 in steps of 25. An analysis of the results follows the graphs (raw image files of graphs may be found in the */testResults/Report Results/Graphs folder*).

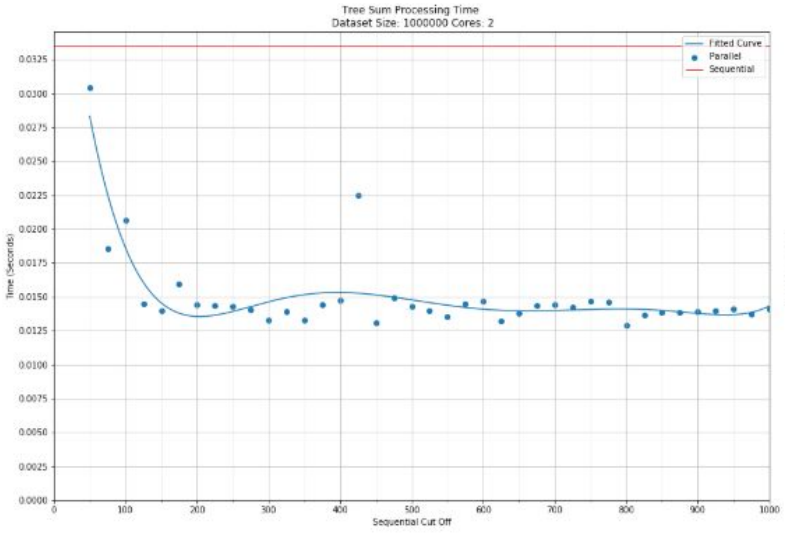


Fig. 7 dataset size: 1'000'000, Cores: 2

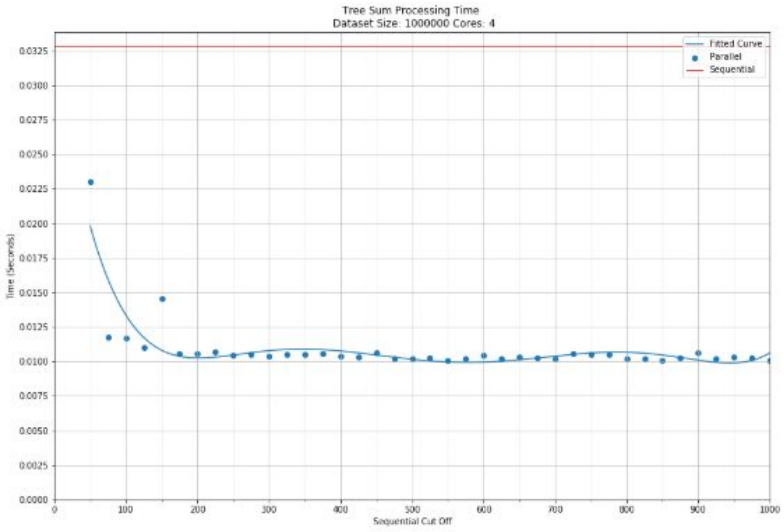


Fig. 8 dataset size: 1'000'000, Cores: 4

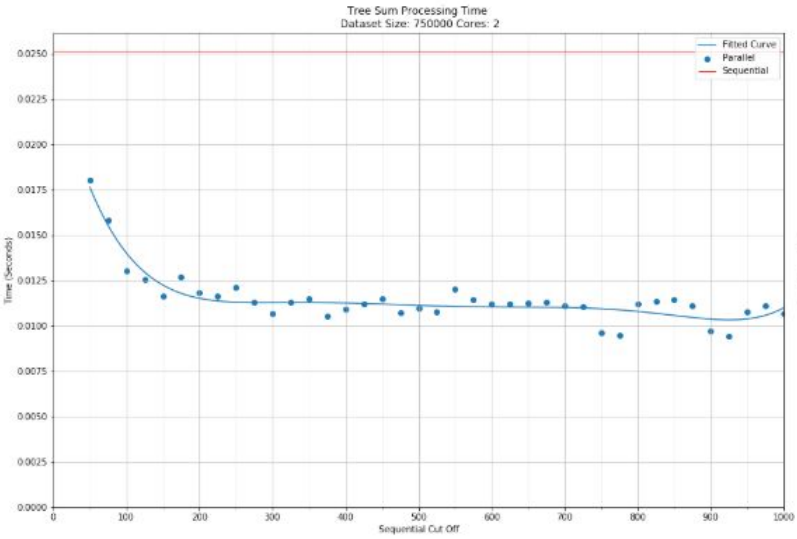


Fig. 9 dataset size: 750’000, Cores: 2

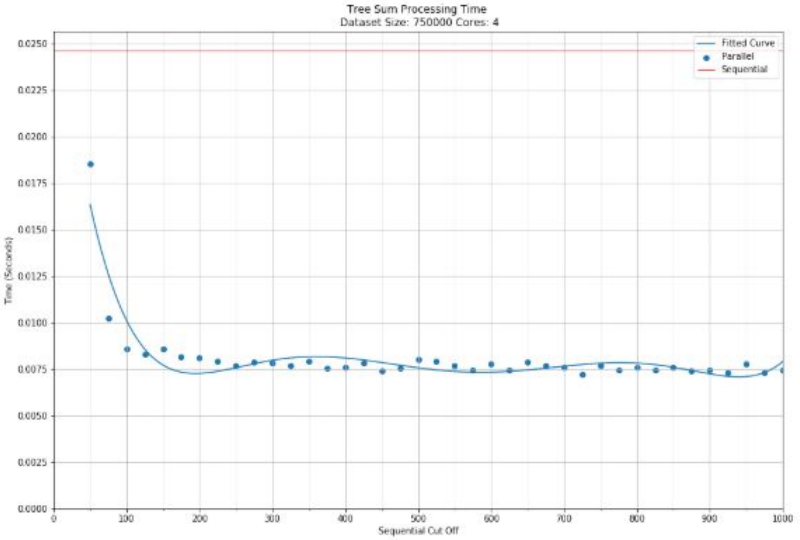


Fig. 10 dataset size: 750’000, Cores: 4

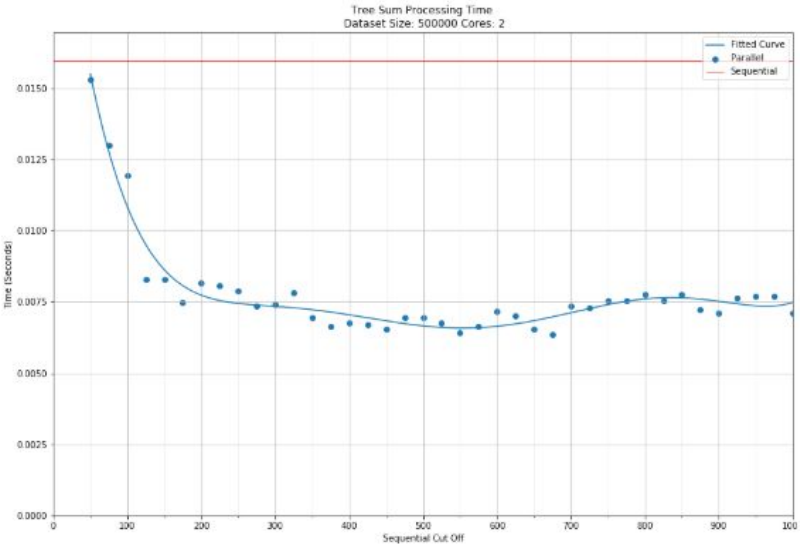


Fig. 11 dataset size: 500’000, Cores: 2

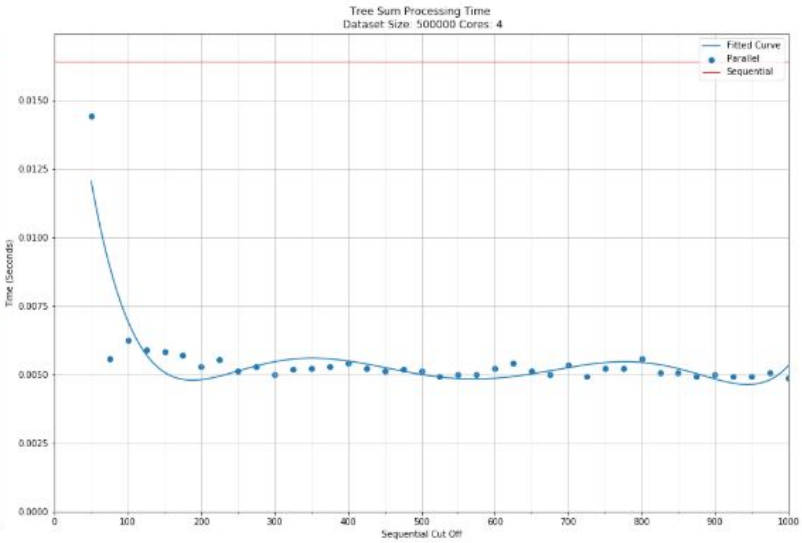


Fig. 12 dataset size: 500’000, Cores: 4

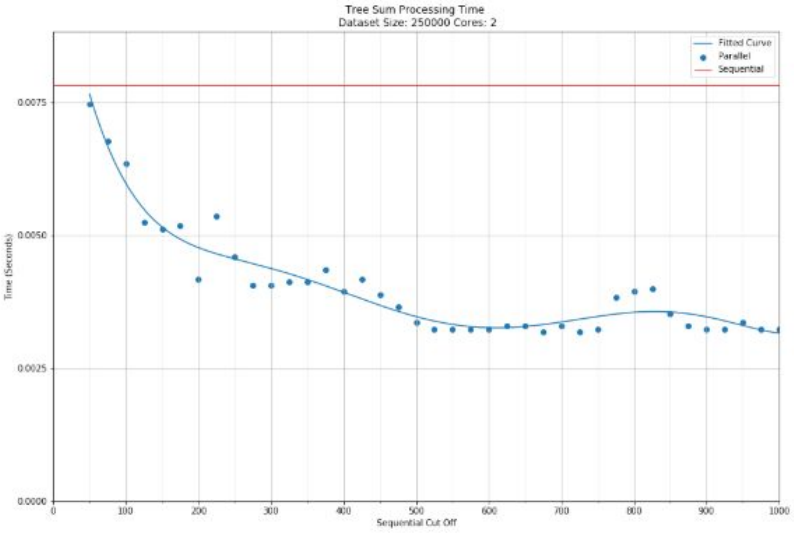


Fig.13 dataset size: 250'000, Cores: 2

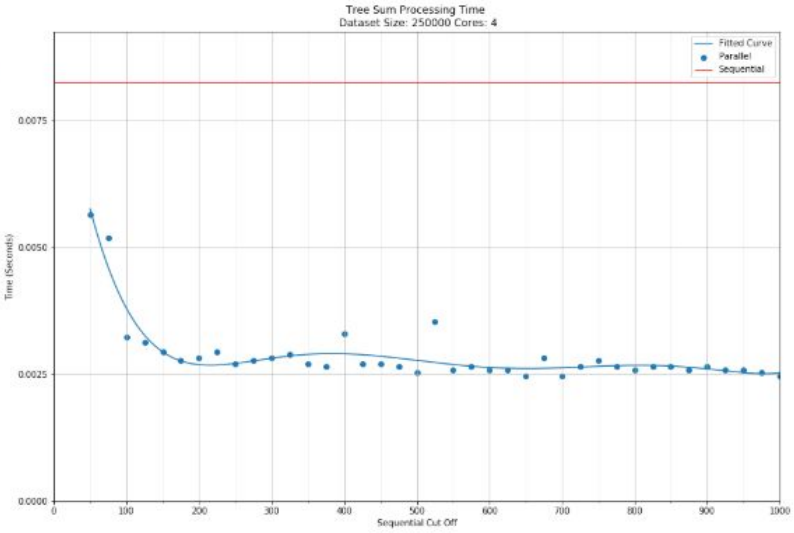


Fig. 14 dataset size: 250'000, Cores: 4

General Observations

Based on the graphical representations of the results above a few observations about general trends and outcomes can be made:

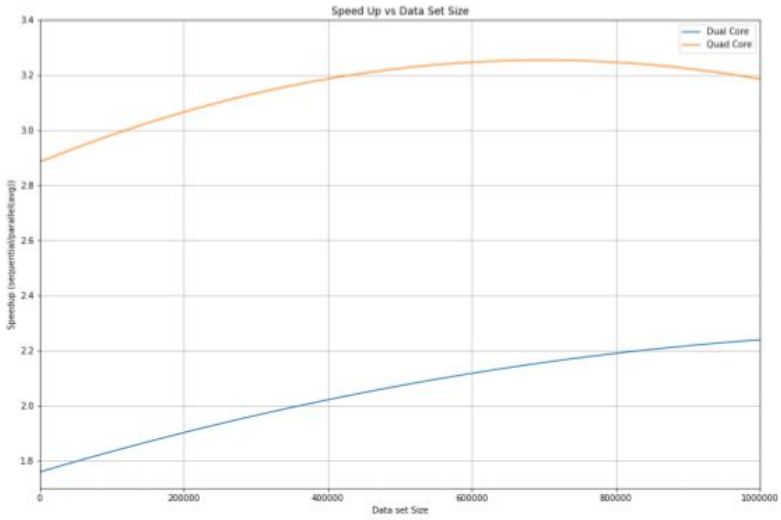
1. The time taken for the sequential algorithm seems to be more or less equal whether the program was run on a machine architecture with two cores or four. This was true across all data set sizes, as seen in **Fig. 8 - Fig. 14**. This matches what we would expect since, by definition, the sequential algorithm operates in one thread only on a single processor regardless of how many processors are available.

2. On average (**Fig. 8 - Fig. 14)** the running time of the parallel algorithm was shorter when the program was run on the quad core machine compared to when it was run on the dual core machine. Also, as the size of the data set increases the gap between the average running time of the two architectures appears to increase slightly.

3. Each result experience a sharp increase in processing time when the value of the sequential cut off approached zero. This spike happened for slightly larger values of the sequential cut off when the program was run on the dual core architecture than on the quad core. These results are also to be expected due to the fact that creating too many threads comes with overhead, which is essentially the cost to processing and memory of creating and destroying threads. This cost is more easily handled on architectures with more processors (capacity).

Measuring Speedup

Using the data from each experiment illustrated in **Fig.8 - Fig.14** the average speedup in performance was calculated by calculating the ratio of the average speed of the sequential algorithm (the time taken for one processor) and the average speed of the parallel algorithm (the time taken for N processors) over an interval of sequential cutoffs, from 500 to 1000. **Fig. 15** shows a table of the speedup ratios calculated for the different data set sizes, with a plotted interpolating quadratic curve going through all the data points.



Speed Up Ratio Table

	Dual Core	Quad Core
250'000	1.94 x	3.10 x
500'000	2.05 x	3.219 x
750'000	2.19 x	3.258 x
1'000'000	2.23 x	3.184 x

Fig.15 Graph representing speedup relative to data set size.

As per the data, the speed up of the parallel algorithm when run on the dual core architecture over the various data set sizes ranges from 1.94x to 2.23x speedup, which roughly matches the calculated expectation. The speed up of the parallel algorithm when run on the quad core architecture over the various data set sizes ranges from 3.10x to 3.258x speedup, which is less that our calculated expectation, but shows that an increase in the number of cores does not produce a linear improvement in processing performance and speed, which is what Amdahl's law illustrates. One very important observation from the data shown in **Fig. 15** is that the parallel programs show significant increases in speedup proportional to the the size of the data (the larger the data set size the better the performance). This was one of the flaws in the logic underpinning Amdahl's law; an increase in the problem size can indeed cause significant speedup in performance algorithms relative to sequential ones because the optimal sequential cut off should remain relatively fixed, while an increase in data size causes the fraction corresponding to the sequential portion of the work, $\frac{Number\ of\ items\ to\ be\ added\ sequentially}{Total\ number\ of\ items}$ in our specific problem, to decrease rapidly especially when dealing with data set sizes $O(10^6)$. This causes an increase to the parallel portion of the work which should logically increase the expected speedup in performance of the parallel algorithm.

Answers to Specific Questions

1. Is it worth using parallelization (multithreading) to tackle this problem in Java?

Yes. Based on the experimental data, using a parallel divide and conquer algorithm for processing large amounts of data as per this specific problem produces a significant speedup in relative performance.

2. For what range of data set sizes does your parallel program perform well?

Larger data set sizes benefit more in terms of speedup in processing performance when running the parallel program. The results on both architectures show that a data set size of 250’000 of above would yield significant speedup in performance, and extrapolating based on the speedup graph in **Fig. 15** it appears a data set size of about 100’000 would yield an increase of about 1.5x, above which point it is arguably worthwhile running a program in parallel.

3. What is the maximum speedup obtainable with your parallel approach?

On the quad core machine, for the a data set of 750 000, a speedup of 3.258x can be achieved, where as a speedup of 2.23x was the maximum achieved on the dual core machine for a data set size of 1’000’000.

4. How close is this speedup to the ideal expected?

The speed up on the dual core machine exceeds initial expectations, where the speedup on the quad core architecture performs worse than expected, but as stated above, based on Amdahl's law a two times increase in processors should not be expected to produce a further 2x increase in processing speed, the speedup expected is more logarithmic as a function on the number of processors used.

5. What is an optimal sequential cutoff for this problem? (Note that the optimal sequential cutoff can vary based on dataset size and architecture.)

ALthough the data above was fairly noisy, it can be argued that there is a trend in the optimal sequential cutoff across different data set sizes and on different architectures. The run times shown as point data in **Fig. 7 to Fig 14** all appear to ‘settle down’ to a particular value of the sequential cut off specific to each problem. Below is a table summary of the observations:

Data size	**Dual Core**	**Quad Core**

250 000	≅ 500	≅ 300
500 000	≅ 400	≅ 300
750 000	≅ 300	≅ 250
1 000 000	≅ 200	≅ 150

As stated previously the optimal sequential cutoff for increases as the size of the data set is decreased. The optimal values in this particular case would then be the highest observed for all data set sizes tested: **500** for the dual core architecture and **300** for the quad core architecture.

6. What is the optimal number of threads on each architecture?

Given the optimal sequential cut offs above the ideal number of threads for each architecture assuming a data set size of 1'000'000 can be calculated as follows:

$$Number\ of\ Threads_{(2\ cores)} = \frac{Total\ Data\ Set\ Size}{sequential\ cut\ off} = \frac{1000000}{500} = 2000\ threads$$

$$Number\ of\ Threads_{(4\ cores)} = \frac{Total\ Data\ Set\ Size}{sequential\ cut\ off} = \frac{1000000}{350} \approx 2857\ threads$$

Conclusions

Based on the results above and the experience of the assignment the following conclusions may be drawn:

1. The theory of parallelism and expected speedup is ideal and does not take into account the peripheral processing work which needs to be done or the specific hardware constraints of particular architectures. Experimental results may vary depending on these factors.

2. Efficient programming can make a big difference to the increase in performance of a parallel program (e.g. the types of operations performed).

3. Parallel performance increases proportionally to the number of processors available as well as the size of the data set. Performance does not increase linearly for either of these, instead increasing logarithmically for each.

4. Sequential programs should run at similar speeds regardless of the number of processors.

5. The optimal sequential cutoff depends on the specific machine architecture used, as well as the size of the data set. It appears to

decrease as the number of processors increase and decrease with an increase in data size.

6. The optimal number of threads are directly related to the sequential cutoff and the size of the data set.