Assignment 1
CSC2002S
MLLDYL002

Introduction

Two main models exist for writing computer programs. These are the sequential and parallel models.

In the sequential model, code and alogrithims are part of a single sequence of instructions that run in step, one after each other on a single processor core. The Parallel model of programming in contrast, allows for multiple segments/sections of code to run at the same time on multiple processor cores.

The aim of this assignment will be to compare the performance of sequential and parallel implementations of a terrain classification algorithim written in Java using the Fork/Join framework. This framework makes use of process threads which, using a divide and conquer technique, fork and combine the computational result of each thread to produce the final result.

Performance will be compared across varying input data sizes and number of threads (sequential cutoff). In general, parallel or multithreaded performance depends on the 'granularity' of the algorithm, input dataset size, processing overhead and the number of cores that all threads are distributed across.

For example, for a relatively large input dataset and low thread per core count it is expected that the processing overhead will dominate and that performance will be comparible to the sequential version (where number of threads is defined by the sequential cutoff parameter) However as the thread count per core increases it is expected that parallel runtime performance will gradually increase until a maximum point of efficiency is reached at which point performance will start decreasing as the overhead to create and combine results becomes increasingly expensive versus the number of results obtained.

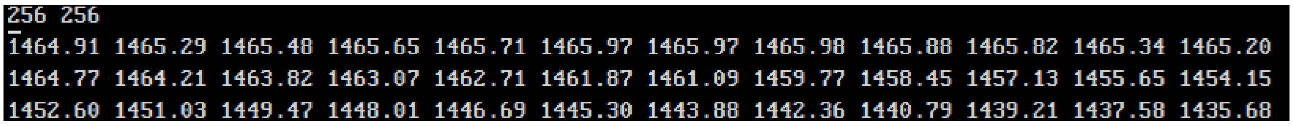
Methods

The assignment involved the implementation of sequential and parallel versions of the terrain classification algorithim. The implementation details of the sequential version will be detailed first followed by the details of the parallel implementation. The sequential and parallel versions are quite similar in some respects, namely the file handling routines and early stage processing blocks share some similarities.

File Processing

For the sequential/parallel implementation it was specified in the project leaflet that two command line arguments be passed/handled by the program. The first for the input file, containing the terrain data to be processed/classified and the second for the corresponding output file, containing the results of the classification algorithm/pass.

Fig.1 shows some sample data from an input file

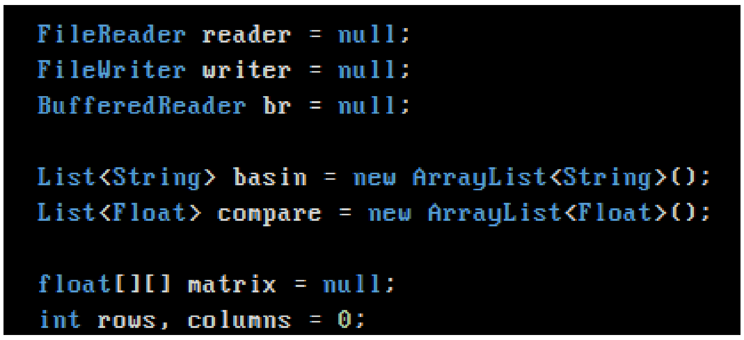


```
256 256
1464.91 1465.29 1465.48 1465.65 1465.71 1465.97 1465.97 1465.98 1465.88 1465.82 1465.34 1465.20
1464.77 1464.21 1463.82 1463.07 1462.71 1461.87 1461.09 1459.77 1458.45 1457.13 1455.65 1454.15
1452.60 1451.03 1449.47 1448.01 1446.69 1445.30 1443.88 1442.36 1440.79 1439.21 1437.58 1435.68
```

Fig.1 - sample input data. (rows, colums at top) followed by data

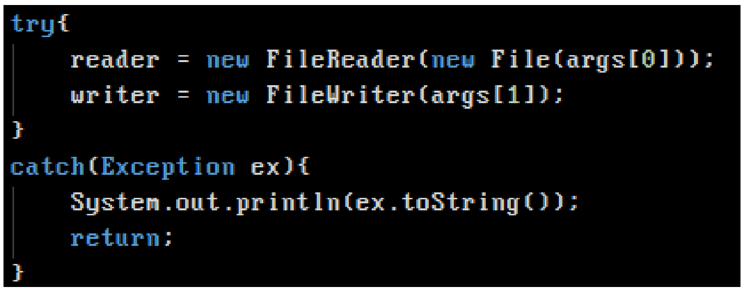
As can be seen from **Fig.1** the file is seen consisting of a single line containing the total number of grid rows and columns followed by a linear array of amplitudes for each of the corresponding grid points.

To handle the reading/writing of files, a FileReader and FileWriter class were utilized within a try-catch block to open both file streams. **Fig.2** shows the general top level definitions within main, while **Fig.3** indicates the try-catch block.



```java
FileReader reader = null;
FileWriter writer = null;
BufferedReader br = null;

List<String> basin = new ArrayList<String>();
List<Float> compare = new ArrayList<Float>();

float[][] matrix = null;
int rows, columns = 0;
```

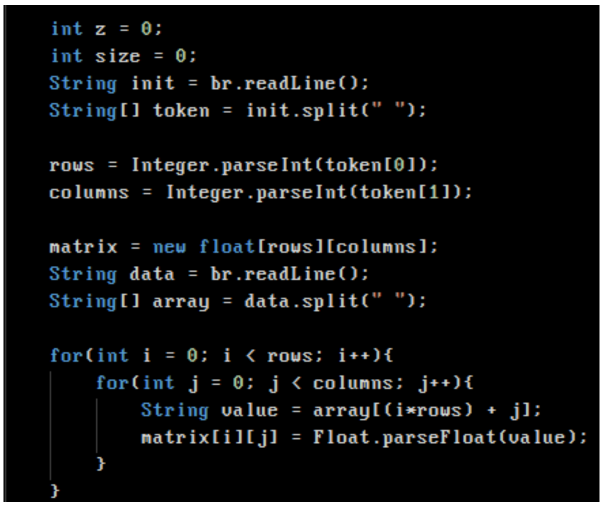
Fig.2 – top level definitions in main function (used for various processing operations)



```java
try{
    reader = new FileReader(new File(args[0]));
    writer = new FileWriter(args[1]);
}
catch(Exception ex){
    System.out.println(ex.toString());
    return;
}
```

Fig.3 - try-catch block for FileReader, FileWriter classes

The rows and columns as found in the top line of the input file (**Fig.1**) are read using the code block shown in **Fig.4.** From this, the grid data (bytes) are then read in, split and placed into a two dimensional floating point array (called matrix) which is then later utilized by the classification algorithm for indexing grid points.

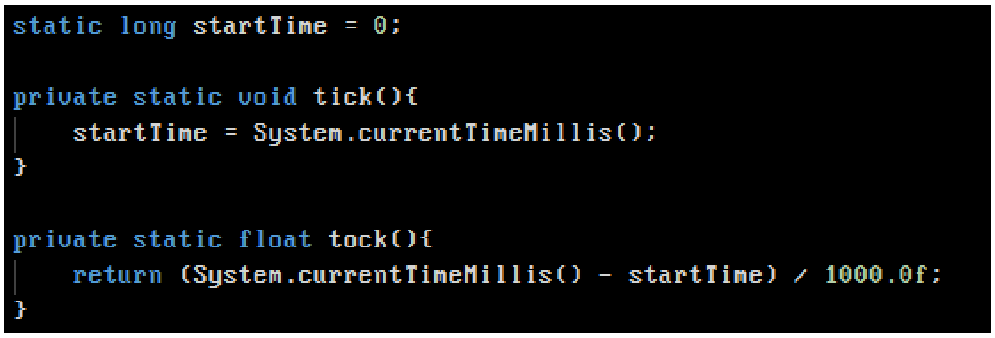


```java
int z = 0;
int size = 0;
String init = br.readLine();
String[] token = init.split(" ");

rows = Integer.parseInt(token[0]);
columns = Integer.parseInt(token[1]);

matrix = new float[rows][columns];
String data = br.readLine();
String[] array = data.split(" ");

for(int i = 0; i < rows; i++){
    for(int j = 0; j < columns; j++){
        String value = array[(i*rows) + j];
        matrix[i][j] = Float.parseFloat(value);
    }
}
```

Fig.4 – Reading in rows and columns, then storing into two dimensional float matrix to be used by the classification algorithm

Timing

To measure the run time of the classification algorithm, two functions were implemented (tock(), tick()), utilizing the System.currentTimeMillis() method and are shown in **Fig.5**. A static variable startTime is declared to store the starting time.



```
static long startTime = 0;

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

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

Fig.5. Timing functions for run time measurement

Measuring the run time of the classification algorithim then involves calling tick() at the start of algorithmic processing and then tock() at the end, which returns the timing difference in milliseconds.

Sequential Classification Algorithm

The goal of the classification algorithm was to determine all the basins within the provided grid array. The rule provided was that for each point on the grid, the surrounding points would be compared and if the difference in all neighbour heights was greater or equal to 0.01m, then that point would be classified as a basin.

For the sequential implementation of the algorithm an ArrayList of type float was declared (as indicated in **Fig.2**) (compare). As well as a String ArrayList to store the resulting basin coordinates (as a string)

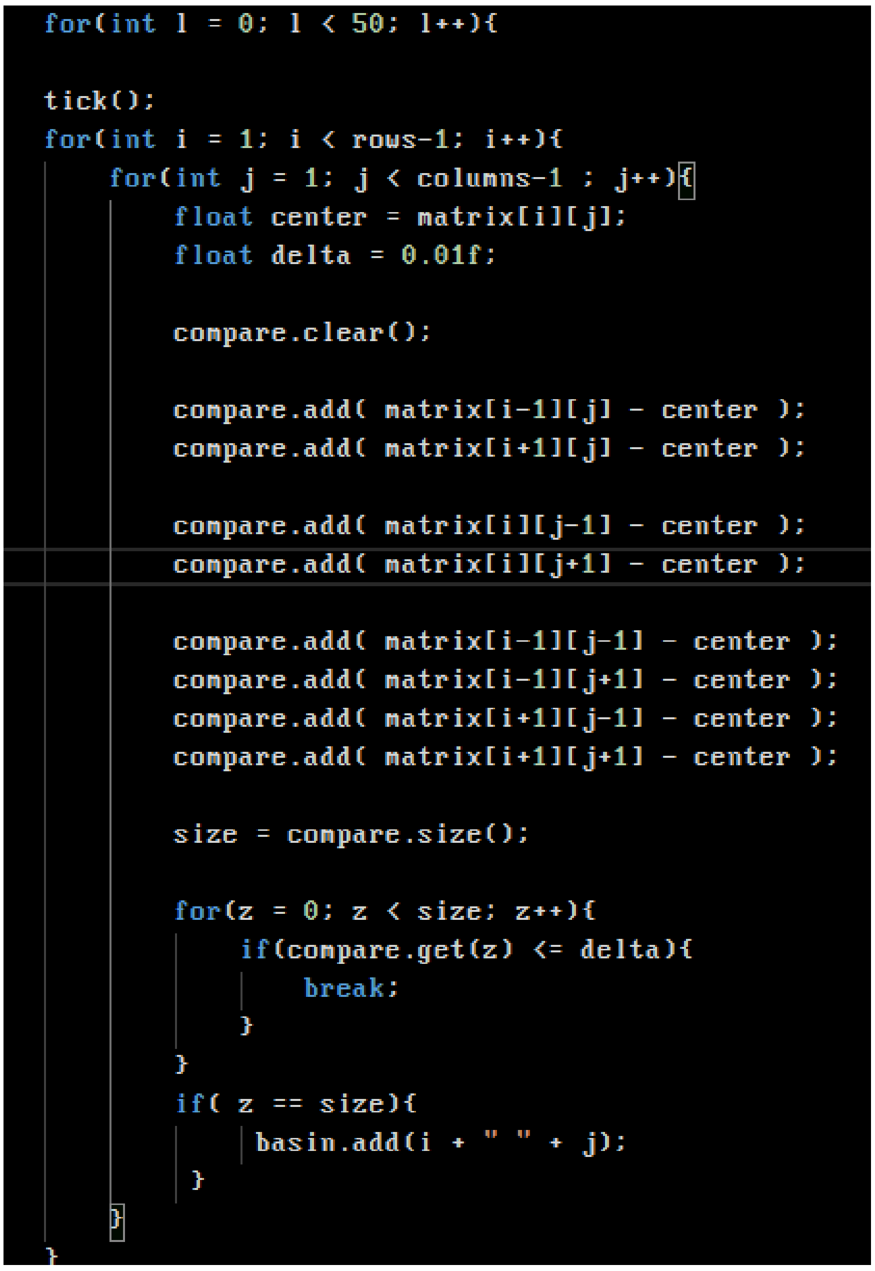
Within a double for loop (to loop over each grid point) 8 difference vectors (differences in the height between the point of interest and specific neighbour value) are added (Fig.6).

The length of each difference vectors is then compared (within a for loop) to check if it is less than the 0.01m threshold, if so, it would not be added to the list of basins.

The classification algorithm is repeated a total of 50 times within a for loop and the average of the last 20 runs calculated. The code block for the sequential classification algorithm is shown in **Fig.6**

** Two float variables are present within the i , j double for loop. The first one **center** holds the height of the current (i , j) grid point that is used for comparison with it’s nearest neighbours. The second float, **delta** holds the comparison value 0.01 in our case.

The left and right matrix elements: [i-1] [j] and [i+1][j] are then added to the compare ArrayList, followed by top and bottom elements: [i][j-1] and [i][j+1] and finally the four diagonals [i-1][j-1], [i-1][j+1], [i+1][j-1] and [i+1][j+1]



```
for(int l = 0; l < 50; l++){

tick();
for(int i = 1; i < rows-1; i++){
    for(int j = 1; j < columns-1 ; j++){
        float center = matrix[i][j];
        float delta = 0.01f;

        compare.clear();

        compare.add( matrix[i-1][j] - center );
        compare.add( matrix[i+1][j] - center );

        compare.add( matrix[i][j-1] - center );
        compare.add( matrix[i][j+1] - center );

        compare.add( matrix[i-1][j-1] - center );
        compare.add( matrix[i-1][j+1] - center );
        compare.add( matrix[i+1][j-1] - center );
        compare.add( matrix[i+1][j+1] - center );

        size = compare.size();

        for(z = 0; z < size; z++){
            if(compare.get(z) <= delta){
                break;
            }
        }
        if( z == size){
            basin.add(i + " " + j);
        }
    }
}
```

Fig.6. Sequential classification algorithm

Parallel Classification Algorithm

The parallel classification algorithm was designed to utilize the concurrent.ForkJoinPool and concurrent.RecursiveTask packages found in Java (Fork/Join). The basic idea was to divide the 2-dimensional float array containing the grid heights, matrix, into multiple sections, defined by an internal constant called ROW_THRESHOLD.

A new RecursiveTask would then be spawned for each corresponding section and a ListArray<String> would be returned containing the locations of the various basins in the format:

row <space> column

The various ArrayList’s would then be appended and used to display the total number as well as each basin coordinate present in the input data set.

For the parallel classification algorithm a class (BasinRecursive) derived from RecursiveTask<List<String>> was constructed. The class was designed around a constructor with the following signature:

BasinRecursive(float[][] **matrix**, int **start**, int **end**)

Here the two-dimensional array, matrix, represents the grid point height data obtained during the file processing stage as indicated in Fig.4

The next two formal parameters, start and end then define the starting and ending row to run the classification algorithm across.

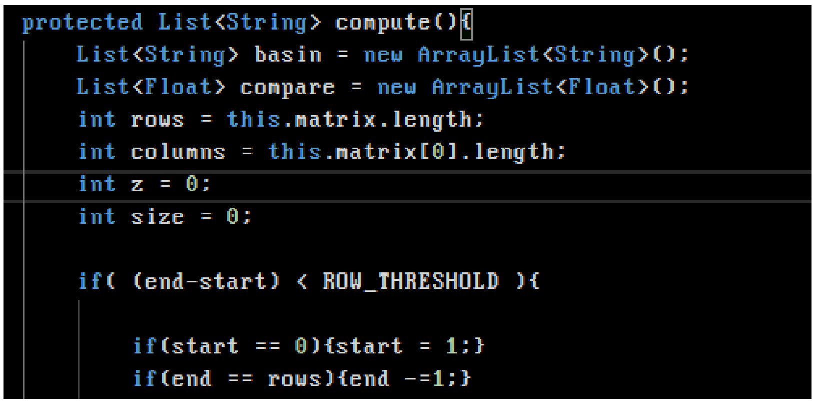
RecursiveTask derived classes are required to implement the compute() class method, which handles the actual work task of the thread.

The compute() method, as found in BasinRecursive, was implemented in a similar fashion to the sequential implementation as shown in Fig.6

However, rather than iterating over the entire matrix float array, only the row region (end-start) defined by the ROW_THRESHOLD would be processed.

Fig.7 shows the parameter declarations as found in compute().
Basin and compare, serve the same purpose in the classification as their sequential counterparts. The rows and columns integer fields simply store the dimensions of the grid point height data (matrix).

A translation code block is also required to ensure that start and end points map correctly



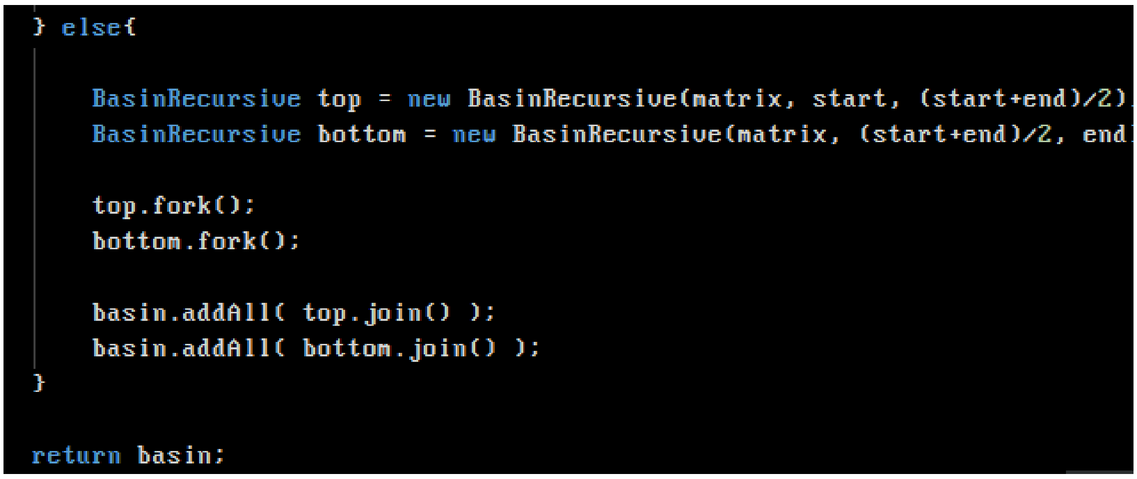
```
protected List<String> compute(){
    List<String> basin = new ArrayList<String>();
    List<Float> compare = new ArrayList<Float>();
    int rows = this.matrix.length;
    int columns = this.matrix[0].length;
    int z = 0;
    int size = 0;

    if( (end-start) < ROW_THRESHOLD ){

        if(start == 0){start = 1;}
        if(end == rows){end -=1;}
```

Fig.7 – compute() parameter declarations

It is worth noting that processing would only occur within compute() if the row difference is less than the ROW_THRESHOLD parameter. If this were not the case then it would be required that the matrix array be further divided using fork() join() calls and this is shown in Fig.8

Initially two BasinRecursive classes would be spawned, with the one containing the top half of the processing matrix and the other containing the bottom half. Essentially dividing the work into half on each interation. A call to fork() on both **top** and **bottom** would then create new subtasks with the results of both top and bottom being added to the basin ArrayList using join()

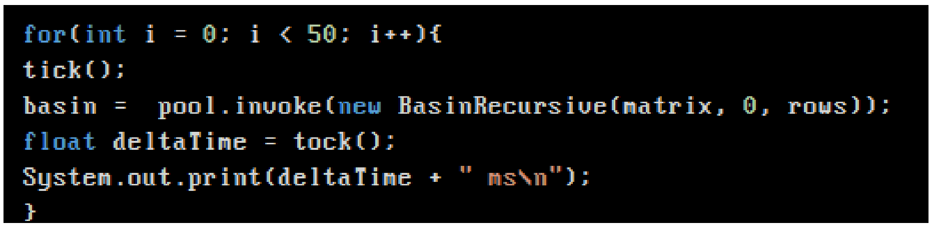
Finally, the result of all the basins is returned for use by the parent class.



```
} else{

    BasinRecursive top = new BasinRecursive(matrix, start, (start+end)/2)
    BasinRecursive bottom = new BasinRecursive(matrix, (start+end)/2, end

    top.fork();
    bottom.fork();

    basin.addAll( top.join() );
    basin.addAll( bottom.join() );
}

return basin;
```

Fig.8 – fork() join() for compute

Within the parent class (BasinCompute) a ForkJoinPool class (pool) was declared and Fig.9 shows the code block responsible for spawning the BasinRecursive child classes as discussed above. This is essentially accomplished through a call to invoke() containing the class definition.



```
for(int i = 0; i < 50; i++){
tick();
basin =  pool.invoke(new BasinRecursive(matrix, 0, rows));
float deltaTime = tock();
System.out.print(deltaTime + " ms\n");
}
```

Fig.9 – Call to invoke

Similar to the sequential implementation, the parallel classification algorithm utilizes the tick() and tock() timing functions as described above.

Time Speedup Measurements & Validation

Since one of the goals of the assignment was to measure the performance/speedup of the parallel implementation over the sequential implementation, a measure of the performance improvement of the parallel classification algorithm versus the sequential implementation was required.

This improvement in run-time performance was calculated as: Ts/Tp. Where Ts is the sequential runtime and Tp is the parallel runtime. This performance ratio was then plotted for various different input dataset sizes as well as various ROW_THRESHOLD values.

For each run of the specific implementation (sequential or parallel) :

java app [file:in] [file:out]

for both the sequential and parallel versions of the classification algorithm, 40/50 of the output values are added to a float ArrayList representing 40 passes of the classification algorithm, with the first 10 passes being ignored. The average of the ArrayList is then calculated and output to STD_OUT.

The method used to plot the speedup of the classification then involved varying specific parameters depending on the value to plot the performance measurement against.

For example when plotting run-time performance vs input data set size, the file:in argument was modified to load in files of varying sizes. For plotting the run-time performance vs number of threads (sequential cutoff) the ROW_THRESHOLD variable was altered.

For each run, the output to STD_OUT was then redirected/appended to another file

Output data was validated by comparing the output data of the parallel implementation , following the classification pass, with the sequential version as well as the provided sample inputs/outputs

Machine Architecture

The computer system used to run both sequential and parallel versions of classification algorithm is described below:

- CPU: AMD Athlon Dual-Core @ 3.8GHz
- RAM: 8GB
- Linux

Difficulties encountered

The CPU used for benchmarking (due to limited resources) did not have many cores (only 2) and therefore difficulties in measuring wide variations in runtime performance of the parallel classification algorithm vs sequential cutoff (ROW_THRESHOLD) were encountered. As mentioned in the assignment leaflet a multicore workstation was preferred, and 4 cores are usually more common and would have provided better results.

Results and Discussion

Firstly a plot of the sequential runtime vs datasize for various input datasizes was plotted and is shown in Fig.10.

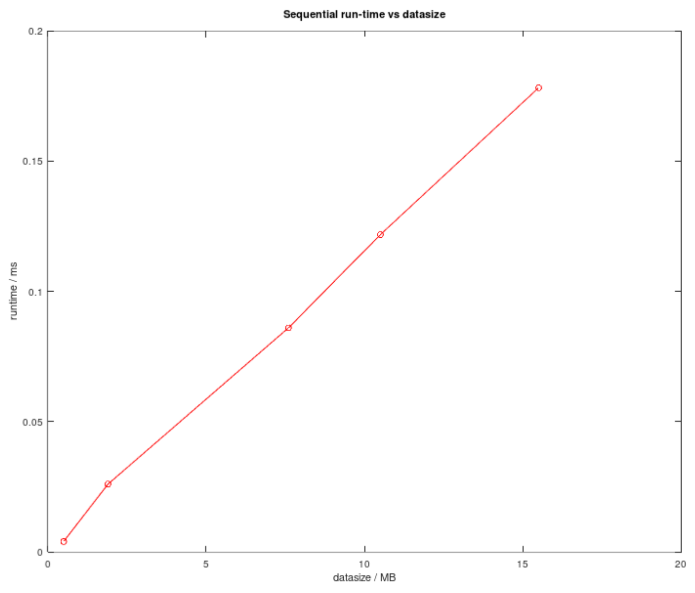


Fig.10 – Sequential run-time vs datasize

Here is can be seen that as the datasize increases, the corresponding sequential runtime follows a linear trend upwards.

For example at a datasize of 1.9MB we get a sequential runtime of 0.026ms, at a datasize of 7.6MB we get a sequential runtime of 0.086ms.

Next a graph of parallel runtime vs sequential cutoff for various datasizes was plotted. Fig.11, Fig.12 and Fig.13 show plots for datasizes (7.6MB, 1.9MB and 500KB respectively)

In all cases a generalized trend is observed, whereby for smaller values of sequential cut-off, the parallel runtime is seen to be minimum. However as the sequential cutoff is increased the run time is seen increasing linearly up to a point at which the run-time is seen leveling off.

It is also worth noting that larger datasets, i.e in Fig.11 (datasize=7.6MB) tend to have greater maximum values then smaller datasets (i.e Fig.12), which is to be expected as the increased number of grid rows/columns increases the processing time, leading to a greater run-time.

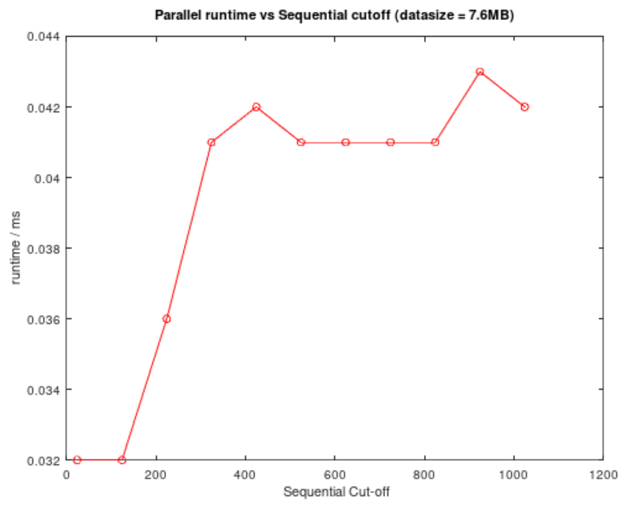


Fig.11 – Parallel runtime vs Sequential cutoff (datasize = 7.6MB)

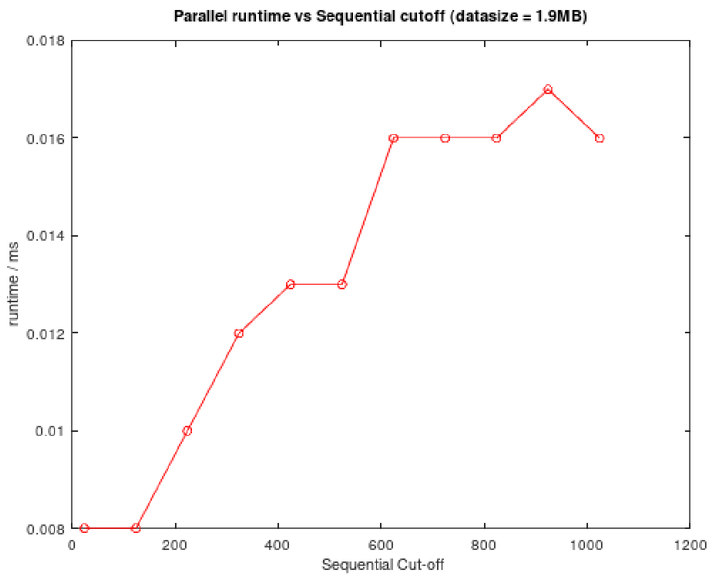


Fig.12 – Parallel runtime vs Sequential cutoff (datasize = 1.9MB

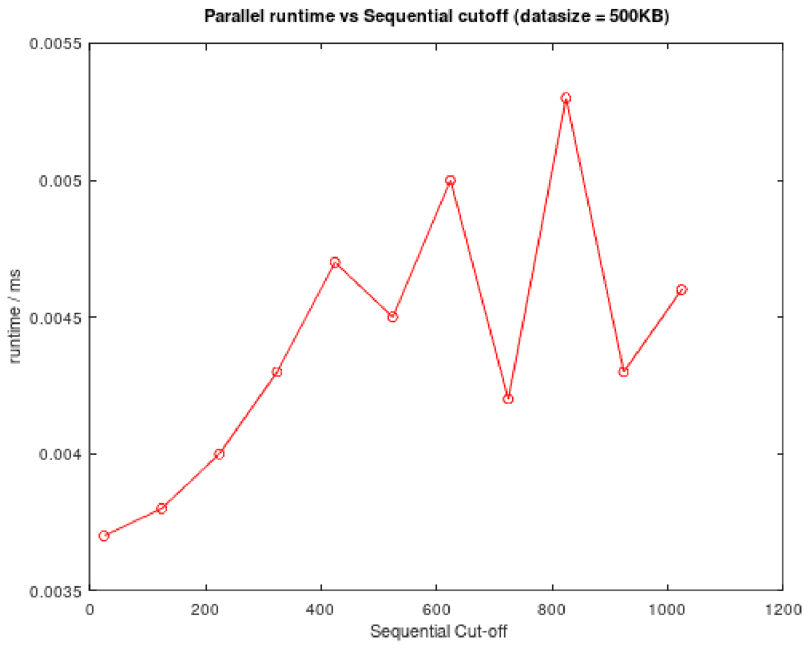


Fig.13 – Parallel runtime vs Sequential cutoff (datasize = 500KB)

To guage the performance difference between parallel and sequential plots the parallel speedup (given by Ts/Tp) was plotted for various data sizes and are indicated in Fig.14, Fig.15, Fig.16.

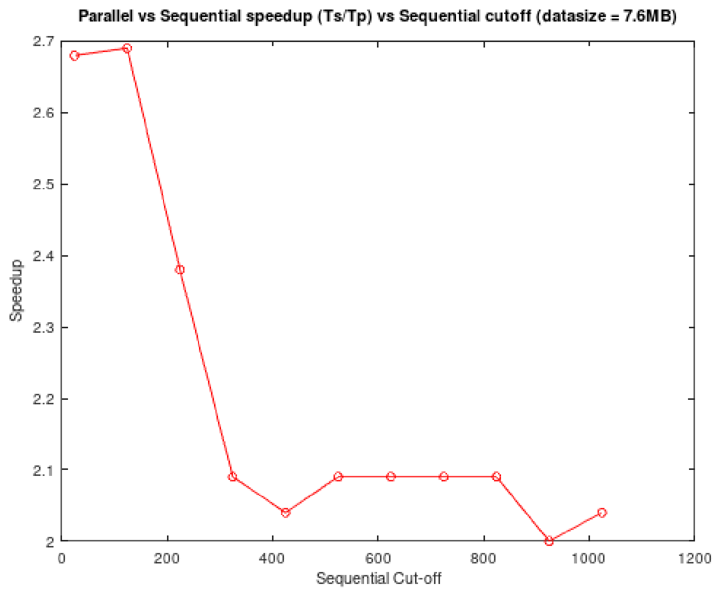


Fig.14 – Parallel vs sequential speedup (Ts/Tp) (fordatasize = 7.6MB)

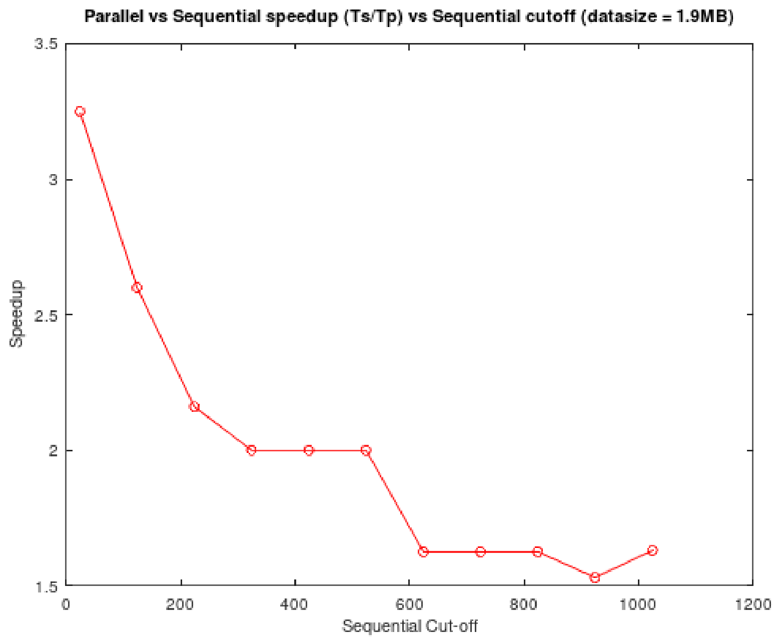


Fig.15 – Parallel vs sequential speedup (Ts/Tp) (for datasize = 1.9MB)

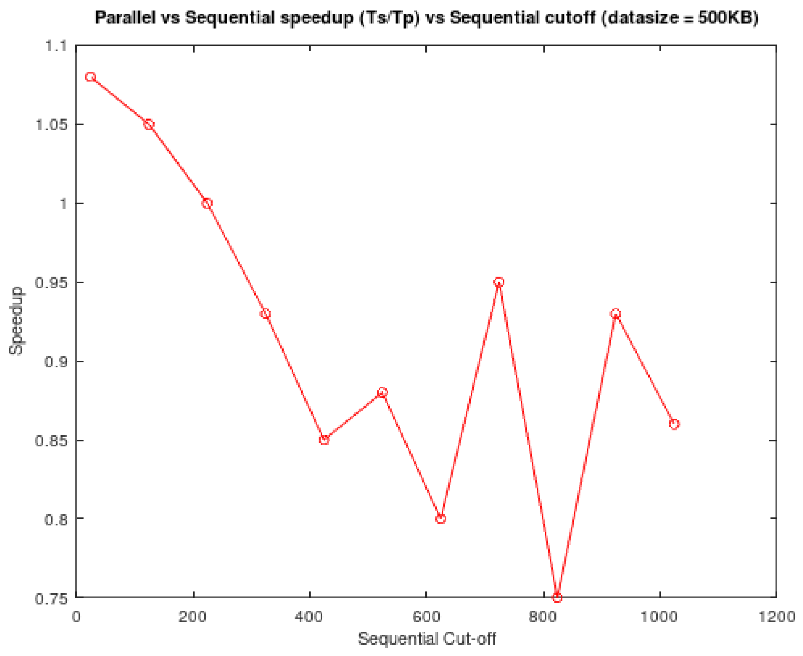


Fig.16 – Parallel vs sequential speedup (Ts/Tp) (for datasize = 500KB)

As can be seen from Fig.14, Fig.15 and Fig.16, speedup is greatest at lower sequential cutoffs and gradually decreases for increasing values of sequential cutoff. The lowest value of sequential cutoff tested was 24 which in all cases resulted in the largest speedup.

Also the maximum value of speedup was seen to vary depending on the input datasize. For Fig.16 the maximum value of speedup was seen to be around 1.1, for Fig.15, the speedup was seen approaching around 3.5, while for Fig.14 the speedup was seen decreasing to around 2.7.

This indicates that there is an optimum datasize at which the speedup is greatest.

For all 3 test cases (datasize’s around 500KB-7.6MB) parallelization did improve the overall performance, which the greatest perfomance increase at a datasize of around 1.9MB.
The optimum sequential cutoff was seen at 24

Conclusions

- Parallelization can be very beneficial on many-core systems (in this assignment I was restricted to 2 cores)

- Overall the parallelization of the terrain classification system using the Fork/Join Java Framework, did improve the performance for the three main datasizes tested: 7.6MB, 1.9MB and 500KB.

- The greatest performance increase/speedup was seen at the lowest sequential cutoff tested (24)

- The efficiency/speedup provided by parallelization depends on the input data-size as well as thread-count

- Parallelization using the Java ForkJoin framework is an effective approach at optomizing sequential algorithms