CSC2002S

PARALLEL PROGRAMMING

September 17, 2018

Student ID: HLLYAS001
Name: Yaseen Hull
University of Cape Town
Department of Computer Science

Contents

1 Introduction **2**

1.1 Java's ForkJoin Framework . . . 3

1.2 Speed up . . . 4

1.3 Problem . . . 5

2 Method **6**

2.1 Creating a tree object . . . 6

2.2 Creating a Parallel Sum . . . 7

2.3 Algorithm validation . . . 8

2.4 Method testing . . . 9

2.5 Problems encountered . . . 9

3 Results and discussion **11**

3.1 Performance comparisons and speedup . . . 11

3.2 Git log . . . 14

4 Conclusion **15**

Chapter 1

Introduction

Parallelism can be described as using extra computational resources to solve a problem faster and more efficiently. It is the division of work into smaller parts shared amongst a pool of threads invoked by the user. In this instance, we implement a data decomposition process in which one problem (array of values) is split over many helper threads which perform the same work on the values. This can be identified as the concept of shared memory. The idea behind shared memory is that threads, which can be thought of as sub processes, share the same address space i.e. they share a collection of static fields and objects. Each thread however has its own local variables, counter and call stack. Threads have access to shared memory and is permitted to read data concurrently, however reading and writing of multiple threads is not allowed due to concurrency issues. To mitigate the non-deterministic nature and concurrency issues while still implementing parallel programming we use java's ForkJoin Framework. The framework allows for a divide and conquer approach in which recursion is used to divide the problem into segments. The key concept is that for a given computation on a data set, e.g. summation of an array, we can divide the array and recursively sum the elements from lower to middle indices as well as recursively sum the values from middle to upper indices. The recursion for both parts uses parallelism to divide the work from their halves in half again. From the two recursive steps we can then add the two to get the full summation. To regulate the thread work, a synchronization primitive known as join is used to coordinate threads. Basically, join was created to prevent a thread from performing any task unless another thread has already been terminated. This also resolves the issue of non-determinism of threads. An interpretation of this idea is illustrated below.

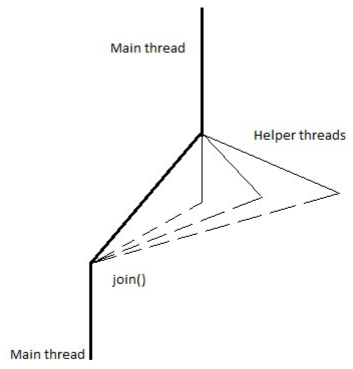


Figure 1.1: Join function

1.1 JAVA'S FORKJOIN FRAMEWORK

The core of the fork/join framework is the ForkJoinPool (pool of threads) class which implements an important work-stealing algorithm and can execute ForkJoinTask processes. The fork/join framework distributes tasks to worker threads in a thread pool which can steal tasks from other threads upon completion of its own tasks. In practice the framework first "forks" the tasks, dividing it up into smaller sub-tasks that are simple enough to be executed asynchronously. The "join" then recursively adds all the sub-tasks together to produce a result.

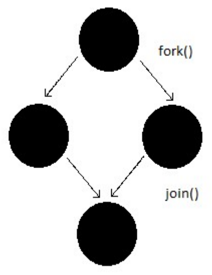


Figure 1.2: Fork and Join

ForkJoinTasks is an Abstract base class for task running within the ForkJoinPool. The sub-classes of the ForkJoinTasks are RecursiveAction and RecursiveTask of which both is extended in practice. Since RecursiveAction and RecursiveTaskis a ForkJoinTask it can be invoked by the ForkJoinPool method ForkJoinPool.invoke() accepting ForkJoinTask as an argument. The difference between the two extensions is that RecursiveAction contains a void compute method whereas RecursiveTask is a generic class with one type parameter in which the compute method must return the type value.

1.2 SPEED UP

In order to calculate the speed up, we need to obtain the running time of a program on one processor and multiple processors. We can define this by getting the ratio of work (time taken for one processor) to span (time taken for an infinite number of processors. Generally, for a divide and conquer algorithm we can generate the work and span by:

Work/ T_1: $3N-2$ with O(n) time complexity
Span/ T_p: $2log_2(N)+1$ with O($log_2(N)$) time complexity

Where N is the maximum number of divisions of tasks (see figure below). Speed up however is found by T_1/T_p and in an ideal situation should have linear scalability. For example, if we have a four-core machine then idealistically we should achieve four times speed up. In our example we have an input array containing one million values. The speed up in a divide and conquer algorithm with a sequential cutoff of 125000 should be:

$N = 1000000/125000$
$N = 8$
$T_1 = 3(8)-2$
$T_1 = 22$
$T_p = 2log_2(8)+1$
$T_p = 7$
$Speedup = 22/7$
$Speedup = 3.1$

Theoretically decreasing the sequential cutoff should increase the speed up drastically. However, parallelism is about decreasing the span (longest path) without increasing the work (number of nodes) too much. The maximum speed is a measure of parallelism in our program although at some point adding more processors won't do anything since it is dependent on the span. Since this is a theoretical value, we can only assume that the actual speed up would be much lower to other constraints on the system architecture and efficiency of code.

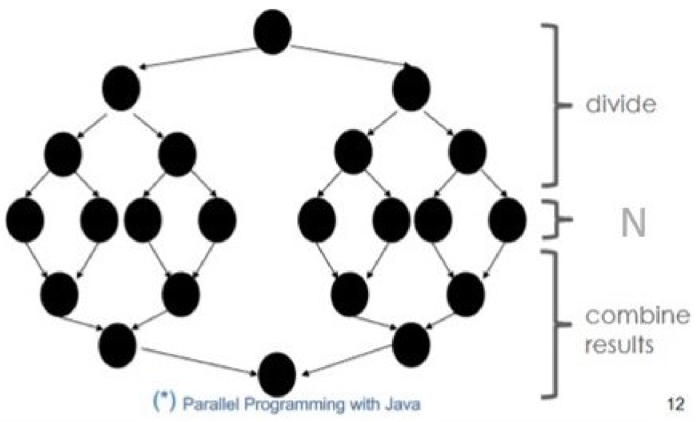


Figure 1.3: Dividing problem

1.3 PROBLEM

Given a 2D dataset of sunlight exposure values over a specified terrain size compute the sum of sunlight exposure for a given tree with specified origin and extent. For a small number of trees this computation might seem trivial if done sequentially. However, processing the summation of tree area exposure for a million trees can be inefficient using sequential means. The same problem can however be done in parallel where a specified number of helper threads perform the same sequential algorithms on a smaller range of trees concurrently. Once each trees area of exposure values have been quantified get the total sun exposure for all trees and the average.
The specifics of this practical thus requires some output array of tree area exposure values to be totalled and averaged. Hence RecursiveTask was chosen for the implementation.

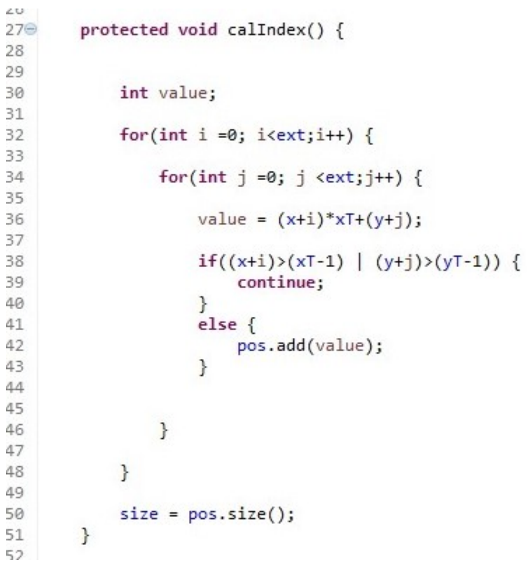
Chapter 2

Method

For the sake of focusing on the problem, an explanation of any reading and writing of input and output has been omitted. Simply following the comments in the code is enough information regarding this topic.

2.1 Creating a tree object

It was essential to the operation of the program that a tree object containing all the relevant data was created. The tree object takes in the terrain x and y dimensions, the tree x and y coordinates as well as the extent of the tree. Using all these variables the coordinates for each element in the array of exposure values was found omitting those falling beyond the terrain boundaries. A general formula for finding the indices of relevant elements in the terrain data array was found using a nested loop (see line 36 below).



```
27      protected void calIndex() {
28
29
30          int value;
31
32          for(int i =0; i<ext;i++) {
33
34              for(int j =0; j <ext;j++) {
35
36                  value = (x+i)*xT+(y+j);
37
38                  if((x+i)>(xT-1) | (y+j)>(yT-1)) {
39                      continue;
40                  }
41                  else {
42                      pos.add(value);
43                  }
44
45
46              }
47
48          }
49
50          size = pos.size();
51      }
```

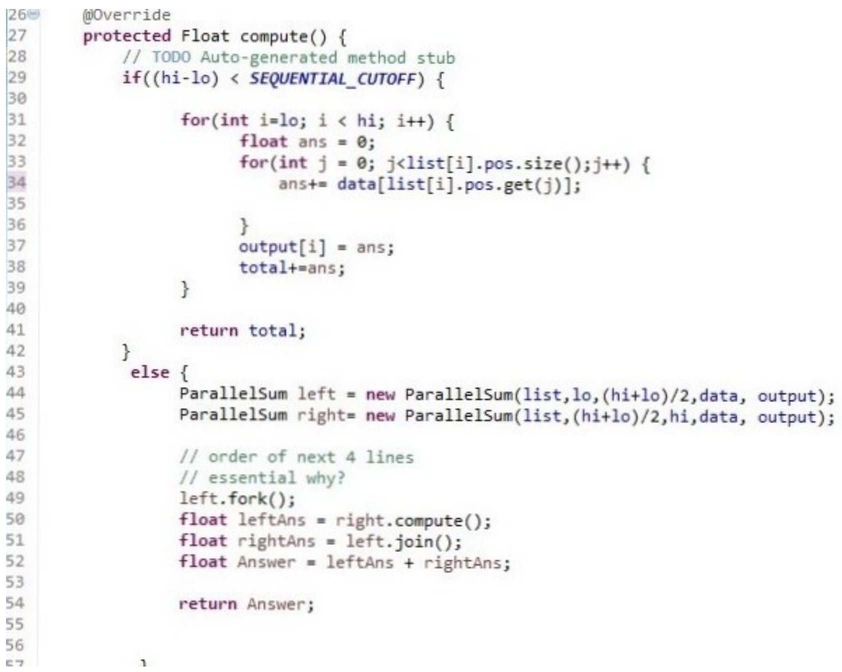
Figure 2.1: Extent calculation

Where
ext =extent
x, y = tree coordinates
xT = terrain size
$value$ = index of elements corresponding to calculated x and y coordinate of tree in terrain data array

The formula for the variable 'value' takes the 2d coordinates of each element in the trees extent and terrain size to obtain the index in the terrain array after which it is added to an ArrayList. The index generation would occur sequentially as a new instance of each tree object was created from the input data.

2.2 Creating a Parallel Sum

As mentioned in chapter one, a sub-class of the ForkJoinTask known as the RecursiveTask class was used to perform the parallel programming. The RecursiveTask is a generic class for which a return type should be specified. In our instance the type would be float as the problem required us to sum float values of trees in parallel. Once invoked, the RecursiveTask of the Parallel sum would initialize the arguments taken in. Passed into the constructor was an array of tree values, the minimum index and length of this array as well as the terrain data array and output array. Chapter one discusses most of how java's ForkJoin framework operates but for clarity's sake a section of the compute method has been extracted below.



```
26      @Override
27      protected Float compute() {
28          // TODO Auto-generated method stub
29          if((hi-lo) < SEQUENTIAL_CUTOFF) {
30
31                  for(int i=lo; i < hi; i++) {
32                          float ans = 0;
33                          for(int j = 0; j<list[i].pos.size();j++) {
34                              ans+= data[list[i].pos.get(j)];
35
36                          }
37                          output[i] = ans;
38                          total+=ans;
39                  }
40
41                  return total;
42          }
43           else {
44                  ParallelSum left = new ParallelSum(list,lo,(hi+lo)/2,data, output);
45                  ParallelSum right= new ParallelSum(list,(hi+lo)/2,hi,data, output);
46
47                  // order of next 4 lines
48                  // essential why?
49                  left.fork();
50                  float leftAns = right.compute();
51                  float rightAns = left.join();
52                  float Answer = leftAns + rightAns;
53
54                  return Answer;
55
56
```

Figure 2.2: Compute method

Depending on the sequential cut off, the array size is recursively halved in parallel until it satisfies the first 'if' statement (see line 29). The array used in this instance is the list of tree objects. Once small enough each array component is operated on by the pool of threads and within each thread things happen sequentially. For each array component several trees are iterated through with the bounds being 'hi' and 'lo'. A nested loop runs from zero to each tree's generated ArrayList size which contains all the indices for the required float values in the terrain data array. Using the 'ArrayList.get()' method each element is fetched and used as an index value in the terrain data array which is subsequently incremented to the local variable 'ans' in the compute method. In addition to returning the sum, it is added to an output array for storage. Upon each return of the compute method, the left and right parts are added to sum up the total of the tree exposure values until a final answer is reached.

2.3 Algorithm validation

The output file provided for this assignment was a sufficient benchmark to check whether the correct data was being generated or not. In addition to the template provided an extra program was created to assert whether each line produced matches the output in text file of answers. Initially smaller sets of data was tested and the equivalent was done in an excel program. With the pre-determined excel data the output generated could was checked.

2.4 METHOD TESTING

As explained in chapter one the max speed up is a measure of the parallel performance. Hence to test the performance and efficiency of parallelism the parameters and conditions were changed to get an idea of what affects the performance in a significant manner. All the tests are done individually and concurrently with other tests to detect change in performance. These tests include architecture changes as well as internal parameter variation. To obtain an accurate result, each test was done five times and averaged. The tests are enumerated below:

1. Variable sequential cut offs
2. Changing the amount of input data
3. Changing Architecture types

For each test the parallel portion is timed using java's 'System.nanoTime()' method. The time initiates before the parallel sum method takes place and ends immediately after so that we can accurately access the parallelism of the program. To obtain the time in the seconds unit division by 10 million is required.
The timing for each test were compared against each other to obtain an account of which architectures and which sequential cut offs works best. The ratio of the speed up of small data sets will also be compared with that of larger and the full data set.
Additionally a program was written to compare the output file to provided sample output. The program matches all tree sum values to determine if any error was made in calculation. Upon error the program will print a message with the relevant line number. To initiate the program navigate to the Assignment1 folder and compile and run the program 'compare.java' in the terminal and input the two files separated by a space to compare.

2.5 PROBLEMS ENCOUNTERED

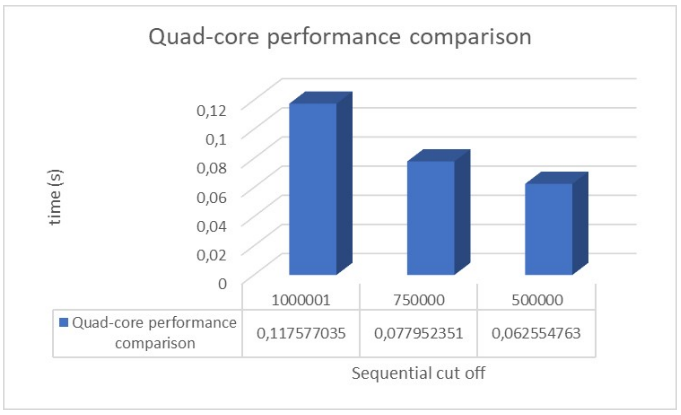
A number of logical problems were encountered initially however through proper interpretation of the problem these issues were resolved. The first being whether to parallelize the terrain data set or the trees data set. It was discovered that writing a program for multiple threads to operate on one tree would be pointless and time consuming. Assuming each tree's elements (x, y and extent) had to be read in sequentially and then using parallelism to find the trees extents would take quite some time considering there are a million tree objects. This was quickly averted by creating an array of tree objects and parallelizing that. The second problem was determining the position of relevant data in the terrain data set. However, treating it like a 2d array was a solution. Hence finding each trees extended x and y coordinates was vital to calculating the index to extract data from the terrain data set. Thirdly there was an issue related to generating indices which were beyond the terrains extent. This took place within the tree objects constructor method but was avoided by a simple if statement which checked whether the coordinates of the tree extents exceeded the terrain boundary. By omitting these values, the ArrayList of indices only included all relevant and required data. Finally getting the correct averaged to the sixth decimal place was troublesome and was not fixed due to time constraints.

Chapter 3

Results and discussion

System Architecture	Quad-core machine	Dual-core machine
cpu	i5-4590	i3-4030u
no. of cores	4	2
no. of threads	4	4
clock rate	3.30ghz	1.9ghz
ram	16gb	4gb
Operating sys.	win10 64bit	win10 64bit

3.1 Performance comparisons and speedup



	1000001	750000	500000
■ Quad-core performance comparison	0,117577035	0,077952351	0,062554763

Figure 3.1: Quad core performance comparison

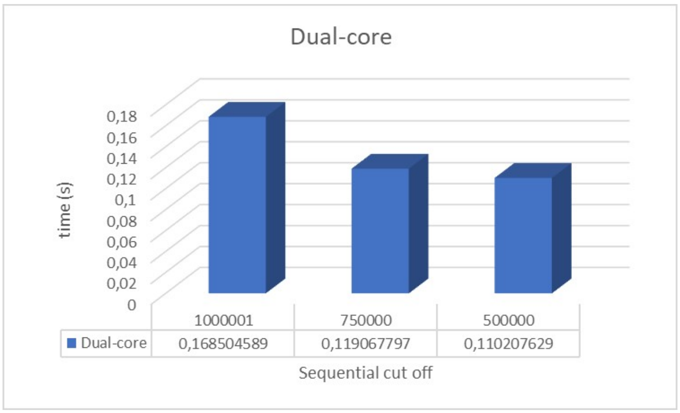


Figure 3.2: Dual core performance comparison

The time taken for each sequential cut off bin was averaged from five consecutive runs. Although only three categories exists in the graph, multiple values was trialed to detect any significant change in time however anything below 500000 was invaluable. It is quite evident that even the sequential performance, bin one, was relatively fast. This could be due to the fact that the objective of the overall problem was simplified by creating tree objects in addition to generating index values for each tree upon reading the input file. The Parallel Sum class only saw a list of tree values for which each trees area sum was calculated and the total of all trees was returned. In terms of speed up parallelism did prove to be useful although not as much as predicted. The best possible speed up for both a Quad-core and Dual-core architecture was:

$Speedup_{quad} = T_1 / T_p$
$Speedup_{quad} = 0.11758/0.06255$
$Speedup_{quad} = 1.87978$

$Speedup_{dual} = T_1 / T_p$
$Speedup_{dual} = 0.16850/0.11021$
$Speedup_{dual} = 1.52890$

As we can see the speedup is rather low, none the less it does increase the processing speed of the program.

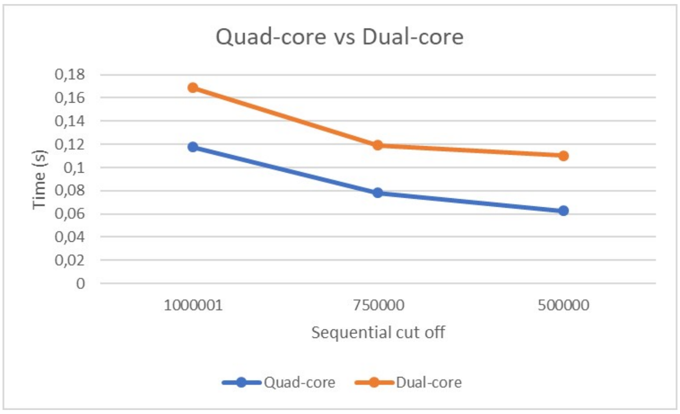
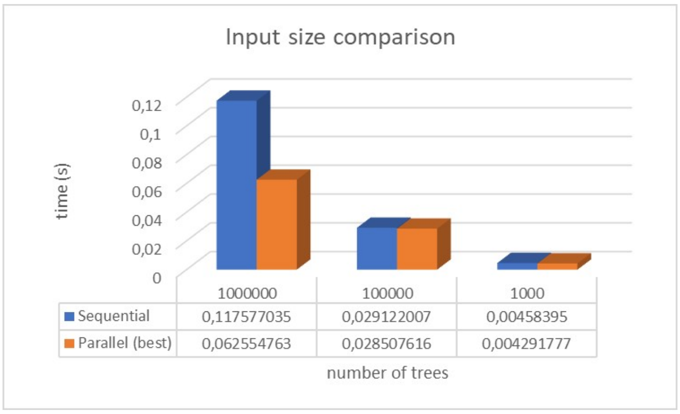


Figure 3.3: Architecture comparison

The performance on different architectures is marginal. Separated by approximately 0. 04 seconds, the dual core machine performed relatively well against the quad core architecture. From the graph it is apparent that the parallel performance for the dual core is mirrored by the quad core keeping the difference more or less constant. This consistency remained even when using smaller sequential cut offs but was left out as smaller cut offs proved to be insignificant.

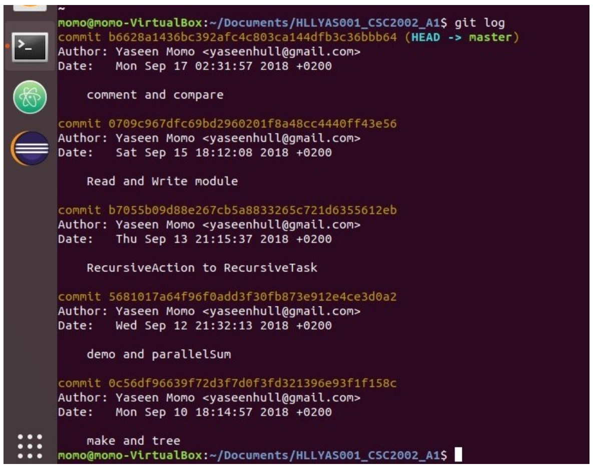


	1000000	100000	1000
■ Sequential	0,117577035	0,029122007	0,00458395
■ Parallel (best)	0,062554763	0,028507616	0,004291777

Figure 3.4: Input comparison

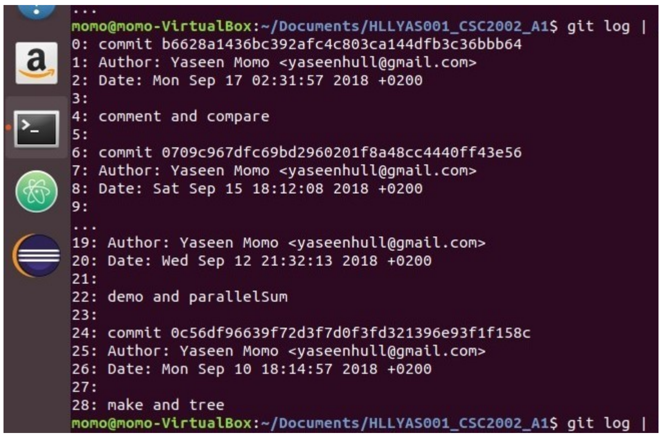
When comparing the different input sizes it is clear that bigger data sets are much more influenced by parallel programming. To compute the speed up for each of the data sets the best possible parallel timing was achieved through using an ideal sequential cut off and the Quad-core architecture. The smaller data sets speed up however is marginally close to one which shows no improvement over using a sequential program.

3.2 GIT LOG



```
~
momo@momo-VirtualBox:~/Documents/HLLYAS001_CSC2002_A1$ git log
commit b6628a1436bc392afc4c803ca144dfb3c36bbb64 (HEAD -> master)
Author: Yaseen Momo <yaseenhull@gmail.com>
Date:   Mon Sep 17 02:31:57 2018 +0200

    comment and compare

commit 0709c967dfc69bd2960201f8a48cc4440ff43e56
Author: Yaseen Momo <yaseenhull@gmail.com>
Date:   Sat Sep 15 18:12:08 2018 +0200

    Read and Write module

commit b7055b09d88e267cb5a8833265c721d6355612eb
Author: Yaseen Momo <yaseenhull@gmail.com>
Date:   Thu Sep 13 21:15:37 2018 +0200

    RecursiveAction to RecursiveTask

commit 5681017a64f96f0add3f30fb873e912e4ce3d0a2
Author: Yaseen Momo <yaseenhull@gmail.com>
Date:   Wed Sep 12 21:32:13 2018 +0200

    demo and parallelSum

commit 0c56df96639f72d3f7d0f3fd321396e93f1f158c
Author: Yaseen Momo <yaseenhull@gmail.com>
Date:   Mon Sep 10 18:14:57 2018 +0200

    make and tree
momo@momo-VirtualBox:~/Documents/HLLYAS001_CSC2002_A1$
```

Figure 3.5: git log



```
...
momo@momo-VirtualBox:~/Documents/HLLYAS001_CSC2002_A1$ git log |
0: commit b6628a1436bc392afc4c803ca144dfb3c36bbb64
1: Author: Yaseen Momo <yaseenhull@gmail.com>
2: Date: Mon Sep 17 02:31:57 2018 +0200
3:
4: comment and compare
5:
6: commit 0709c967dfc69bd2960201f8a48cc4440ff43e56
7: Author: Yaseen Momo <yaseenhull@gmail.com>
8: Date: Sat Sep 15 18:12:08 2018 +0200
9:
...
19: Author: Yaseen Momo <yaseenhull@gmail.com>
20: Date: Wed Sep 12 21:32:13 2018 +0200
21:
22: demo and parallelSum
23:
24: commit 0c56df96639f72d3f7d0f3fd321396e93f1f158c
25: Author: Yaseen Momo <yaseenhull@gmail.com>
26: Date: Mon Sep 10 18:14:57 2018 +0200
27:
28: make and tree
momo@momo-VirtualBox:~/Documents/HLLYAS001_CSC2002_A1$ git log |
```

Figure 3.6: git submissions

The git submissions for the first and last ten commits, although there's only five in total.

Chapter 4

Conclusion

Writing the program was relatively easy however the decision on which data set to use for parallelism posed a problem initially. Once it was apparent that the tree exposure data was to be summed a more intuitive approach was taken and thus the use of the RecursiveTask class was taken. The results obtained communicate that parallel computing is useful and in this instance works. However the speed up for both architectures, different sequential cut offs and input sizes says that not too much of a significant outcome was achieved as the ratio of sequential to parallel programming was very close to one. In terms of architecture, the quad core machine definitely had the upper hand as more cores allows for extra resources to be used concurrently. Decreasing the sequential cut off was also useful up to a certain extent. This could be due to the fact that to achieve good parallelism involves increasing the span while not increasing the work too much. The reason lower sequential cut offs works is due to work being forked off to multiple threads for operation. Finally from the different data sets it is evident that large data sets works well with parallel programming and is deemed fit for this concept of divide and conquer. Thus to improve the results a larger data set should've been used to assess the parallelism and speed up more finely. In addition to this I would run the program on multiple architectures as well as different operating systems for example mac OS. Using a different Operating system could be more useful in terms of timing as windows operates with many background programs in progress.