University of Cape Town

Department of Computer Science

CSC20002S- Assignment 1 report (Parallel programming)

1. Introduction

Traditionally, computer software has been designed as a series of instructions where an instruction is executed one after the other. In situations where time is important, this approach to software development can resort to be ineffective. Computer Manufacturers in 1970s attempted to solve this problem by increasing the number of processors in a machine every two years. This observation is what is called Moore's Law, and was made by Gordon Moore in 1965. However, in late 1990s computers reached what is called power wall: This means that increasing more transistor or clock speed generates so much power loss that cooling mechanisms were impossible to implement.

Processor developers then resorted to developing multi-core processors instead of increasing clock speeds. Parallelism is a kind of computing that uses extra computational resources to solve the problem. A parallel algorithm **must** be **faster** than equivalent serial algorithm and **correct**. In this assignment, the performance of parallelism is made on Monte Carlo algorithm. Monte Carlo methods use probabilistic approach to solve complex numerical and mathematical problems. They are methods that aim to solve two statistical problems, namely **integration** and **optimization**. In this assignment, and Monte Carlo algorithm is used to find the minimum of a two-dimensional function f(x,y) within a specified domain, an optimization problem.

2. Method

A domain of two-dimensional function is represented by a two-dimensional grid of discrete evenly spaced points. Because the grid size can be very large, Monte Carlo algorithm uses probabilistic approach to finding the minimum of function without having to evaluate all points in the grid. This is done by series of searches. For each search, a starting grid point is selected at random and the function is evaluated at this point. The search then attempts to move "downhill" by evaluating the function at neighbouring four grid points and move to the one with lowest value. The search then tries to move down hill again and continues this behaviour until a global minimum has been found.

2.1 Serial Approach

The function in this assignment is considered as a representing the height a terrain, and the program's task is to find the minimum height. The package **MonteCarloMini** defined for this problem contains five classes : **TerrainArea.java, Search.java, MonteCarloMinimization.java,**

MonteCarloMinimizationParallel.java and **DataCollector.java.** A TerrainArea object represents a two-dimensional grid that is searched by a Search object. Moreover, MonteCarloMinimization is a object that implements the serial solution to the problem. It defines the main function which takes arguments in the for <rows> <cols> <xmin> <xmax> <ymin> <ymax> <search density> <mode> <parallel>

<row> and <cols> are integers that represent the number of rows and columns of the grid, respectively. <xmin>,<xmax> represent start and ending point on the x-axis and <ymin> <ymax> represents the boundaries on the y-axis. Furthermore, <search density> is a double that determines the number of searches, <mode> determines whether or not to run the program in **debug mode** and <parallel> determines in parallel computation should included.

The serial approach is very straight forward. The number of search is calculated by the formula: **rows*cols*search_density,** An array of that size is created and traversed with a for loop to search the TerrainArea one search after the other.

2.2 Parallel Approach

Parallel approach uses the same search objects and as used in serial algorithm, but breaks the array recursively using Fork Join framework.

The approach is as follows:

First the array of search objects is divided in half. The left half is forked to an independent thread, while the right half gets computed by the current thread. This process is performed recursively until he array size is within specified threshold or sequential cut off. The search objects within that range are then used to search the TerrainArea and the one with minimum value is returned. This is outlined in the following steps:

Step 1: Create a MonteCarloMinimizationParallel object with original search objects. The constructor invoked below set the upper and lower bounds of that thread to search array length and zero, respectively.

```
ForkJoinPool pool= ForkJoinPool.commonPool();
MonteCarloMinimizationParallel original= new MonteCarloMinimizationParallel(searches);
ArrayList<Double> result= pool.invoke(original);
```

Figure 1: Screenshot showing creation of the main thread that is added in the pool

Step 2: Break the array in half, give the first half to the other array and compute the second half in the current thread.

```
//Thread that search the TerrainArea using Search object on the first half
MonteCarloMinimizationParallel thread1= new MonteCarloMinimizationParallel(startX,mid);
   //Thread that search the TerrainArea using Search object on the second half
MonteCarloMinimizationParallel thread2= new MonteCarloMinimizationParallel(mid,endX);
//give first half to other thread to computer
   thread1.fork();
   ArrayList<Double> here=thread2.compute();//compute the other half in this thread
```

Figure 2: Screen shot showing how the array is divided into halves,recursively

Step 3: when array size is within set threshold, use search object in that portion of the array to search the terrain area, sequentially.

```
    int local_min=Integer.MAX_VALUE;
    int finder =-1;
    for  (int i=startX;i<endX;i++) {
        searchers[i].reset();//reset the step to that it can be used by parallel Algorithm
        local_min=searchers[i].find_valleys();
        if((!searchers[i].isStopped())&&(local_min<min))
        { //don't look at  those who stopped because hit existing path
            min=local_min;
            finder=i; //keep track of who found it
        }
```

Figure 3: Screen shot showing how each thread searched the terrain area sequentially when array size was within the set threshold

Step 4: Wait for the left thread to find and return the one which found the minimum value between the current thread and the thread forked to compute with the left portion of the array.

```
        ArrayList<Double> other=thread1.join();
        //check which found the minimum and return it
        if(here.get(0)<other.get(0)){
            return here;
        }
        return other;
```

Figure 4: Screen shot showing the how waiting for thread was done, and how minimum value was figured out

It should be noted that, the parallel approach used RecursiveTask because each thread needed to return the minimum value of the function it found as well as where that point is, the ArrayList<Double> returned contains this information from each thread.

2.3 Benchmarking

To benchmark both algorithms, timing were bad specifically for Terrain area search and not any other operations in the code. For serial timing, the time are start just fore the for loop the loops through a search array and stop immediately when the loop exits. Similarly for parallel algorithm, timing was started just before creation of ForkJoin pool and stopped immediately returning from the method *ForkJoinPool.invoke(pool).*

With this timing, the main method of class MonteCarloMinimization was invoked in class DataCollector class for testing are against multiple inputs. The first set of input used was variable search densities while all other parameters remain constant. This means that algorithms were timed for different search densities while fixing grid size. This procedure was repeated with varying grid(number of rows and columns), and with variable sequential cut-offs on different machines.

Discussion

The computers that were used and quard-core and octa-core intel processors. The data collected were then analysed in python and the figure below shows the results of the experiment.

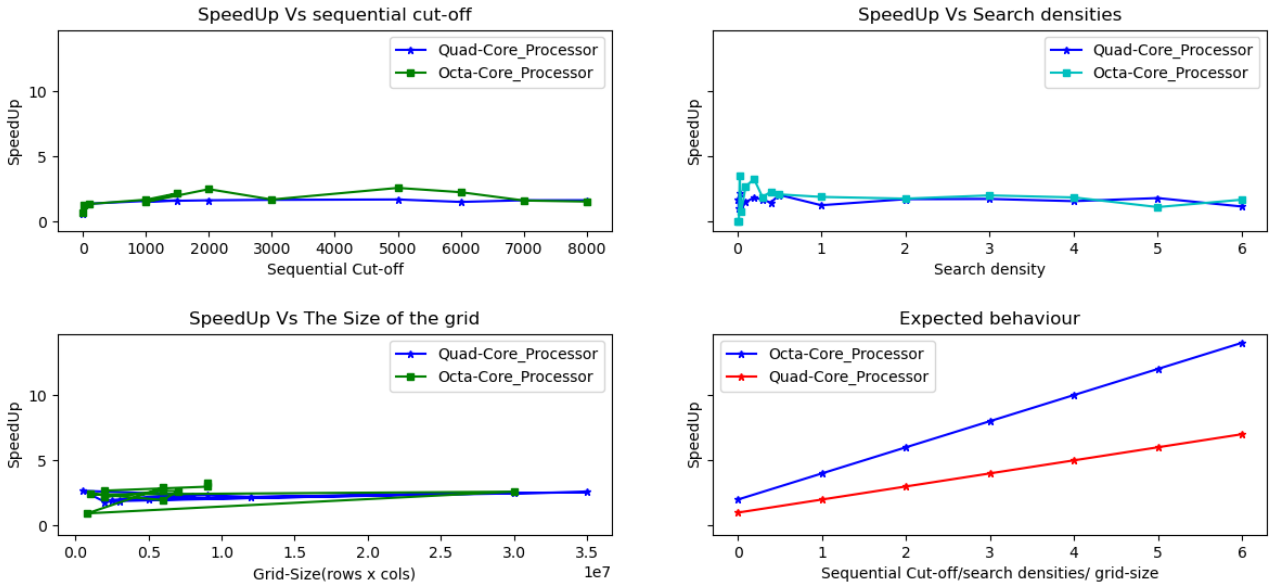

Figure 5: Results of the parallel performance investigation

On figure 5 above, the top left plot is the plotting of the speed against varying sequential cut-off. This plot was produced to visualise how are parallel algorithms influenced by the set threshold. Moreover, the plot of top left corner of fig.5 above is plot of the speed up against search density and the figure of the bottom left corner is the plot of the speed up against grid-size(rows*cols). Lastly, the graph on the bottom right corner show the expected behaviour of the performance of the algorithms when benchmark for sequential-cut-off, search search density and grid-size.

As seen from the figure above, the results, obtained are different from what was expected. Also, the graphs have unexpected spikes. To analyse the result, understanding of what speed up is is crucial. In simple terms, speed up is the ratio of the time taken by serial algorithm to the time taken by parallel algorithm. This means that, a small speed up(less than 1) shows that parallel takes longer than serial algorithm, while high speed up mean that parallel algorithm is taken less time to compute than serial.

From the sequential-cut-off plot above (left top graph), it can be seen that the best speed up around 2000 for 8-core machine, while is almost constant for and quad-core processor. This means that with fewer cores, the parallel algorithm will perform almost the same the serial one, because by definition, parallelism uses **extra** computational resources to solve problem. As seen from the graphs, the plots produced differ significantly from what was expected. The reason behind could be

cause the amount of work was not large enough to parallelise it. The one main requirement of parallelism is that the program must have huge enough work that can be performed in parallel. This also explains why there are some spikes in the plots. Because thread execution time is completely non-deterministic, it is possible that s large data set execute longer than a slightly less data set, creating the sharp decrease in speed up.

Conclusion

Conclusions that can be drawn from the results of the experiment are that parallelism indeed depends on the problem size and sequential cut-off. For the data set used in this experiment, which was limited mainly the heap (RAM) of the computers where test were made, was not large enough for parallelism to worth it. Additionally, It is concluded that for the date set used, which produced no more than 10 million search due to storage constraint, the best speed up is when the sequential cut-off is between 1000 and 2000 search for 8 core processor.

Git Repository Commit History

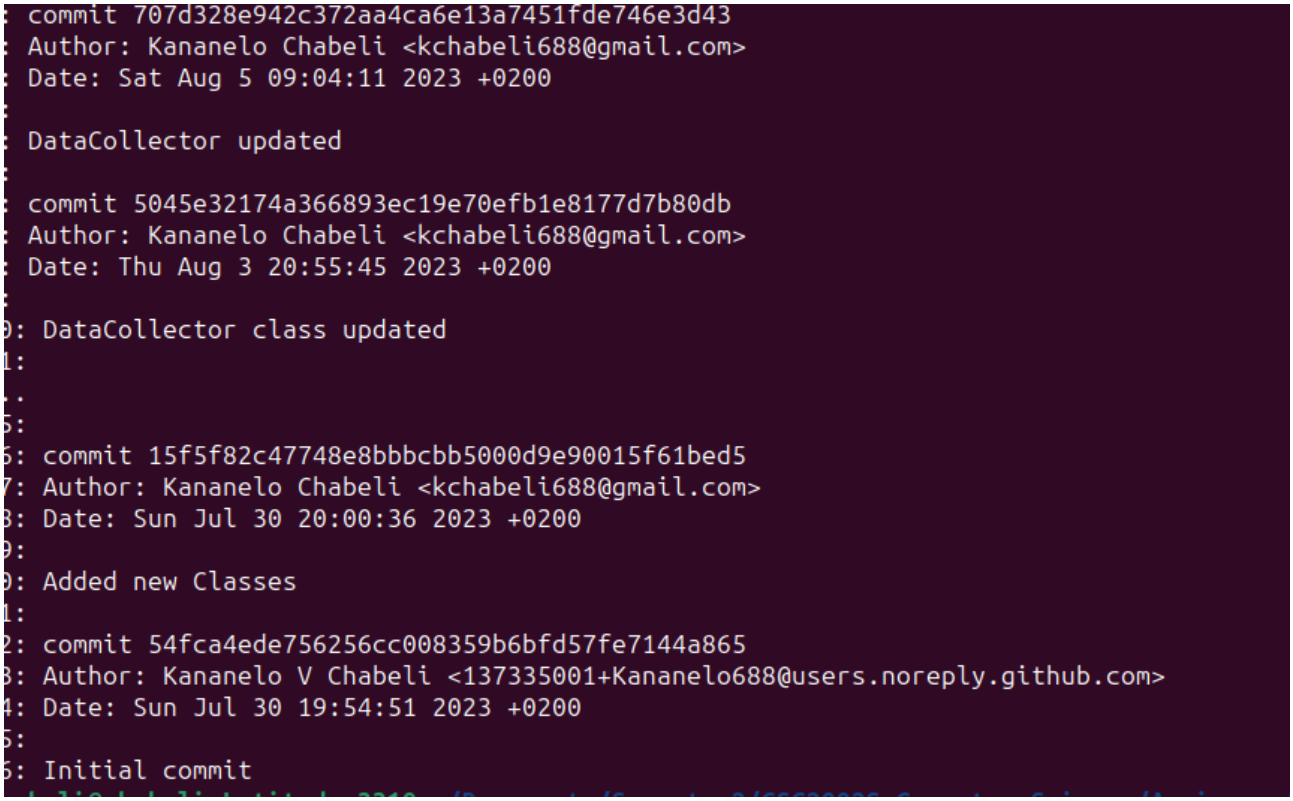

Figure 6: Screenshot showing first 10 and last 10 commits of the repository