RMXDAN002 PCP Assignment 1

Dan Rom

Parallel and Concurrent Programming Assignment 1
CSC2002S

August 2023

0.1 Introduction

This objective of this assignment is to parallelize a Monte Carlo minimization algorithm using the Java Fork/Join framework. The given serial algorithm finds the minimum height of a terrain over a discrete grid. It makes use of three classes - *TerrainArea.java*, *Search.java* and the main class, *MonteCarloMinimization.java*.

The serial program runs as follows: A terrain is created with a function f(x,y) describing the height of the terrain at the given x and y coordinates. The area of the terrain being searched is specified through user-inputted arguments xmin, xmax, ymin and ymax. A discrete grid over this region is specified by two other user-inputted arguments rows and columns. The final user-inputted argument is search density which describes the number of searches per grid point.

A Search object performs a search by evaluating the height function of the terrain at a grid position. The heights of the terrain at all neighbouring grid positions are then evaluated and the search moves to the grid position with the lowest height. This is repeated until the search lands on a position whose neighbouring points all have a higher height than the current position (a local minimum), or until the search lands on a position that has already been evaluated.

An array of Search objects is created - with the total amount of search objects being equal to *rows*columns*search density.* For each element in the array, a random starting grid position is chosen. The searches are executed serially, and all local minimum heights are compared to find the lowest height (the global minimum). The global minimum is then printed to the screen, along with the terrain coordinates of this minimum height as well as the time it took for the entire search.

0.2 Method

0.2.1 Parallelization Approach

The approach used to parallelize the Monte Carlo minimization algorithm involves recursively breaking an array of Search objects up into smaller sections. The maximum number of search objects in each section of the array is determined by setting a sequential cutoff. The value of the sequential cutoff is determined experimentally by observing which values yield the fastest run time. The value used in this parallel implementation is 1000 - therefore there are 1000 search objects in each section of the array.

The searches in each 'smaller section' of the array are executed serially. However, the serial execution of each 'smaller section' of the array is run in parallel to the serial searches of the other sections of the array. Each section of the array returns the minimum height found in that section, as well as the position in the array holding the search that found this minimum. Each minimum returned is compared to find the lowest height (the global minimum).The Java Fork/Join framework is used to achieve the parallel execution of each section in the array. The goal of this is to reduce the time taken for the full search to be run, especially in instances where there is an extremely large number of searches that need to be executed.

One of the issues faced during this process arose due to a constraint on the assignment submission stating that the Search.java class could not be submitted. This proved to be a problem as the Search class could no longer be used to create an array of Search objects. This issue was resolved by creating an inner class, *SearchInner* which contains all fields and methods of the original *Search.java* class. An array of *SearchInner* objects is used to hold all the searches as opposed to the previous array of *Search* objects. *SearchInner* was placed inside the *TerrainArea.java* class as this resulted in the fastest run time.

0.2.2 Validation of Algorithm

Once the parallel approach is implemented, it is important to validate the algorithm and ensure that it is running as expected i.e. returning the correct minimum height of the terrain as well as the correct coordinates. This validation is achieved using two methods.

Method 1: Comparing results to Serial Program

The first method used to validate the parallel implementation involved comparing the results given by the parallel version of the program with those given by the serial version provided both versions of the program are given the same arguments. This method confirmed the minimum height given by the parallel program is correct, however it became difficult to validate the coordinates of this height as they occasionally differ from the coordinates given by the serial program. This could either be due to an error in the program, or due to the terrain having more than one global minimum at different coordinates.

Method 2: Utilizing the Rosenbrock function

This method allows the coordinates given by the parallel program to be validated. The function describing the height of the terrain is changed to the Rosenbrock function. The Rosenbrock function has only one global minimum - a height of 0 at coordinates (1,1). The parallel program is run various times with different valid arguments - each time returning the correct minimum height of 0 at coordinates (1,1). This confirms that the coordinates given by the parallel version of the program are correct.

0.2.3 Benchmarking

The parallel implementation of this Monte Carlo Minimization algorithm is validated. However, it is important to check that the parallel algorithm is worth implementing. Therefore, the algorithm is benchmarked against the serial algorithm in order to determine whether it speeds up the search process. This benchmarking is performed by running the parallel and serial versions of the program with the same arguments and noting the time it takes for both programs to execute all their searches. The speedup of the parallel program is then calculated through the formula: tSerial / tParallel where tSerial is the time taken for the serial program to execute its searches and tParallel is the time taken for the parallel program to do the same.

In order to obtain accurate results, and accommodate for the effects of cache warming, each program is run three times with the same argument. The median of these 3 run times is then recorded.

The arguments used for benchmarking are varied - increasing the grid size, and therefore the total number of searches, each time. The speedup of each run is then plotted against the number of searches executed in that run to provide an overview of the parallel vs serial performance.

Seeing as the speedup is heavily dependant on the number of cores of the machine running the code, two different speedup graphs are created - one where the code is running on a machine with an intel core i5 (4 cores) and the other where it's running on UCT's nightmare server.

0.3 Results

After benchmarking the parallel implementation against the serial version of the program on two different machines, the speedups of each run were recorded and plotted against the number of searches being performed by the program. The speedup graphs can be seen below.

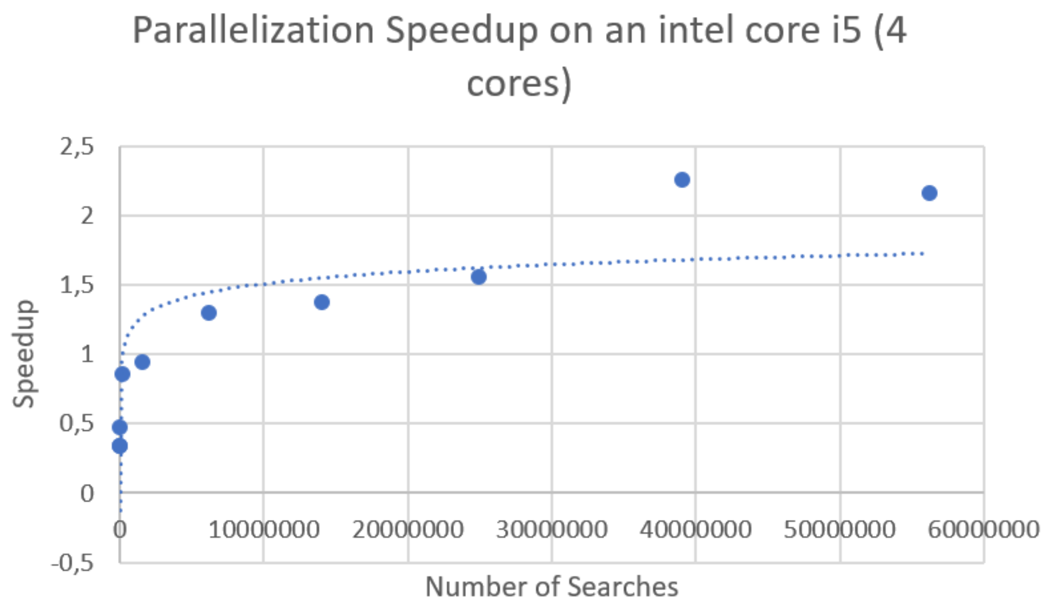

Figure 1: Benchmarking results on intel core i5

As can be seen in the above graph (Figure 1), the parallel program only begins outperforming the serial program at 6,250,000 searches. Before this point, it runs considerably slower than the serial version. This is due to the overhead cost of the time it takes for the program to recursively split up the SearchInner array and assign each section of the array to different threads.

However, as the searches become more and more numerous, this overhead becomes a smaller percentage of the overall run time and the time saved by executing the searches in parallel begins to outweigh the time 'wasted' by splitting the array and creating the threads.

The parallel program running on the i5 reaches a maximum speedup of 2.26 when there are 39,062,500 searches being executed. This corresponds to a grid size of 12500x12500 at a search density of 0.25. It is surprising that the speedup of the program at 39,062,500 searches was slightly higher than the speedup of 2.16 at 56,250,000 searches (corresponding to a grid size of 15000x15000 at a search density of 0.25). It is possible that this can be attributed to the sequential cutoff value chosen (1000). This cutoff may be more suited to a smaller amount of searches and at a certain number of searches, say 56,250,000, it may be more beneficial to have a larger cutoff to reduce the overhead of creating more threads than necessary.

The intel core i5 has a theoretical maximum speedup of 4 (seeing as it has 4 cores which can each execute their own threads at the same time). However, the program only reaches a maximum speedup of 2.26 when running on this CPU. The reasons for this include the overhead of parallelization, as mentioned above. It could also be due to the fact that not every aspect of the program can be parallelized. Many parts of the program are still running sequentially. Another potential reason is that there could be background tasks running on the machine which are using some of the cores. These factors explain the reason why the maximum theoretical speedup of 4 is not reached.

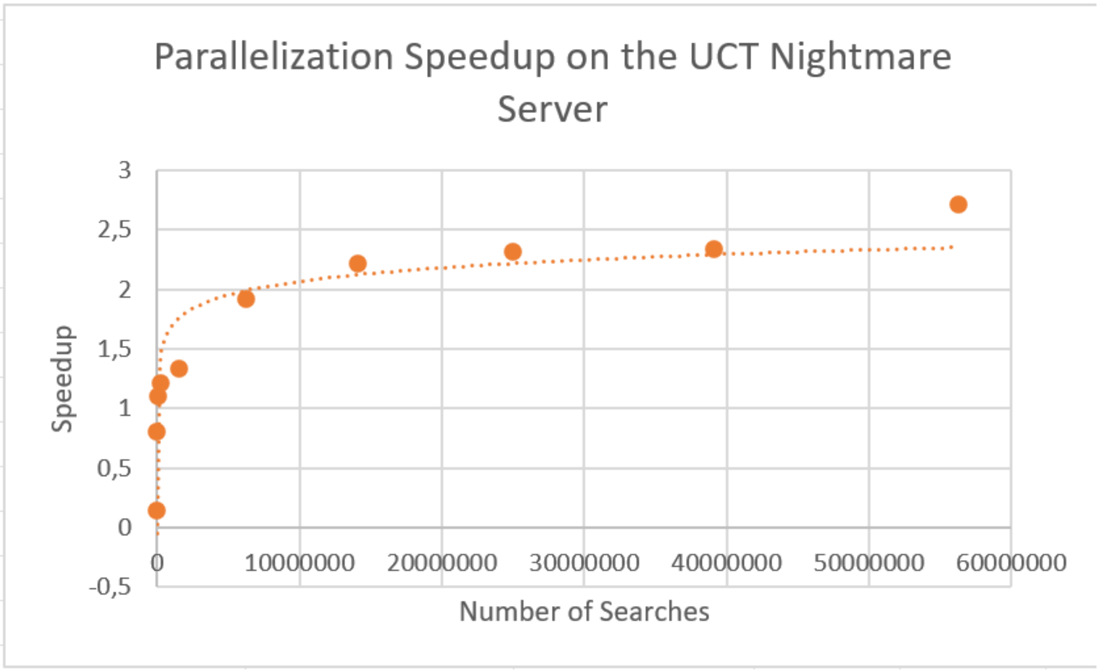

Figure 2: Benchmarking results on UCT's nightmare server

As can be seen in Figure 2 above, the speedup graph of the program running on UCT's nightmare server follows a very similar pattern to that of Figure 1. However, the program ran much more efficiently on the nightmare server - outperforming the serial program at only 62500 searches (corresponding to a grid size of 500x500 at a search density of 0.25).

The program also reaches a higher maximum speedup on the nightmare server compared to the intel core i5 (2.72). This could be due to fewer background tasks running on the nightmare server as opposed to on the machine using the intel core i5. The maximum speedups are very similar on the two different machines as both of them have a total of 4 cores.

0.4 Conclusions

By referring back to the objective of this assignment - parallelizing the Monte Carlo minimization method and seeing if it is worth parallelizing, it can be seen that this assignment was a success. The program was succesfully paralellized and its results validated. It can also be seen by examining Figure 1 and Figure 2, that the program was worth parallelizing. The parallel version of the program outperforms the serial version at a reasonable number of searches when run on an intel core i5 as well as on UCT's nightmare server. In fact, in both of these, the parallel version of the program ran more than twice as fast as the serial version at a high enough number of searches.