Melissa Githinji **Due 13/08/23**

CSC2002S Assignment 1

Parallelizing Monte Carlo Function Optimisation

Characterising the serial solution

Getting an idea of the expectations of the parallel solution

The purpose of the serial algorithm is to find the lowest point of a two-dimensional mathematical function, which is visualised as a grid, within a specified range. Finding the lowest point of a function means finding the minimum height in the terrain of the grid. Monte Carlo methods have been implemented to randomly search the grid a given number of times and carry out multiple comparisons of the value of the function to find the smallest.

There is a shared TerrainArea object between multiple Search objects which represents each calculation on the terrain. Each Search accesses a grid point, evaluates the value/height of four neighbouring grid points, compares to the latest minimum, and moves to the direction of the lowest value's gridpoint. This is effectively moving downhill along the gradient of a slope to get to the global minimum. This aligns with the theory of optimization as for simple 1D functions it is known to find the derivative as the gradient and find it's 0 (lowest) value. The Searches are processed one after the other, thus making the algorithm serial.

Methods

Parallelization Approach

Approaching the parallel solution, I decided to use the Fork/Join Framework in Java. This is because the nature of the Monte Carlo method is brute force repetitive calculations. The grid is a static structure and so search operations do not need to interact and can be carried out only needing this level of synchronisation.

Fork/Join is a divide and conquer recursive solution which is the best way to deal with a large quantity of searches quickly. The work is split in half for the threads in the pool to carry on dividing until calculations are carried out, comparisons between the halves are carried out, and the answer is delivered back to the Manager class MonteCarloMinimization.java as an array of the two integers, 'min' and 'finder'. All calculations were restricted to the SearchParallel class in order to keep the code as reusable as possible for testing purposes so that parameters need only be changed in one location.

Problems encountered

Implementing the Fork/Join with multiple threads accessing so many Search objects introduced the Java OutOfMemoryError: Heap Space exception. This exception occurs rightfully for many reasons in this program due to the high memory requirements of storing the searches. For example: JVM Memory is limited and through testing I got to know that having a few hundred million searches is not possible for the parallel solution due to the slightly higher memory requirements over the serial solution.

Validating the algorithm

Using the Rosenbrock function:

The Rosenbrock function describes a surface with a long, narrow, parabolic shaped valley. Since it is known to be difficult to converge to the global minimum of this function, it will put the parallel algorithm to the test.

$$f(x, y) = (1 - x)^2 + 100\left(y - x^2\right)^2$$

The function has a global minimum of 0 at (x,y) = (1,1). Below the serial solution achieves this result.

```
Serial solution:
Run parameters
        Rows: 1000, Columns: 9000
        x: [0,000000, 100,000000], y: [0,000000, 100,000000]
        Search density: 0,300000 (2700000 searches)
Time: 621 ms
Grid points visited: 2764740  (31%)
Grid points evaluated: 13635968  (152%)
Global minimum: 0 at x=1,0 y=1,0
```

The parallel solution finds it as well, thus validating the algorithm's correctness.

```
Parallel solution:
Run parameters
        Rows: 1000, Columns: 9000
        x: [0,000000, 100,000000], y: [0,000000, 100,000000]
        Search density: 0,300000 (2700000 searches)
Time: 274 ms
Grid points visited: 1718549  (19%)
Grid points evaluated: 3335986  (37%)
Global minimum: 0 at x=1,0 y=1,0
```

Benchmarking the algorithm

Machine Architectures Tested:

1) My laptop is MacOS Big Sur 2.2GHz Quad-Core Intel Core i7 with 1 processor. Virtual box is installed which is assigned 2 of the cores for the Linux virtual machine. It has

16 GB 1600 MHz DDR3 RAM and 3 levels of cache with 256 KB L2 Cache per core. The computer is 8-9 years old and has had no hardware replacements/improvements/repairs so slightly less that optimal performance is expected due to 'wear and tear' so to speak.

2) Nightmare has 8 Intel(R) Xeon(R) CPU E5620 cores at 2.40GHz and 48GB RAM. The cores are shared by all users making a connection to the server with 2 threads allocated per core. It has a 64-bit CPU and operating system with a 86 bit instruction set.

Supplement: Boundaries

Now that the algorithm is correct, the input parameters can be varied to study how strongly the solution holds. As alluded in my parallelisation approach, this included finding the benchmarks through trial and error.

For MacOS
I fixed the following: x: [1,000000, 100,000000], y: [1,000000, 100,000000]

1) 1000 x 923498 grid with search density 0.8
   Grid size too big; throws OutOfMemoryError when initialising terrain
   Next test: Decrease grid size
2) 1000 x 123498 grid with search density 0.8
   Too many searches; throws OutOfMemoryError when initialising searches
   Next test: Decrease search density
3) 1000 x 123498 grid with search density 0.2
   Successful and quick
   Next test: increase search density to range (0.2,0.8)
4) 1000 x 123498 grid with search density 0.6
   Too many searches; throws OutOfMemoryError when initialising searches
   Next test: Increase search density to range (0.2,0.6)
5) 1000 x 123498 grid with search density 0.5
   Too many searches; throws OutOfMemoryError when initialising searches
   Next test: Increase search density to range (0.2,0.5)
6) 1000 x 123498 grid with search density 0.3
   Too many searches; throws OutOfMemoryError when initialising searches
   Conclude testing, no search density to 1 decimal place in the range (0.2,0.3)

Therefore, I can conclude that test 3) is the best my algorithm can do on the MacOS architecture due to memory restrictions. Applying the formula for the number of searches on the parameters of test 3) the maximum number of searches the parallel algorithm can perform is:

```
num_searches = (int)( rows * columns * searches_density );
```

= 1000 x 123498 x 0.2
= 24 699 600 searches

(This is an approximate test because I kept search density to 1 decimal place, so I have not found the exact maximum number of searches possible. My computer, however, shall not be subjected to any more intense testing)

Results

Efficiency is a measure of time increase; therefore, speedup was recorded as the ratio of the time taken for the serial solution vs the parallel solution. The mass of the calculations, or the essential operation, is evaluating a search, therefore ***problem size is number of searchers****. And grid size is directly proportional to number of searches.*

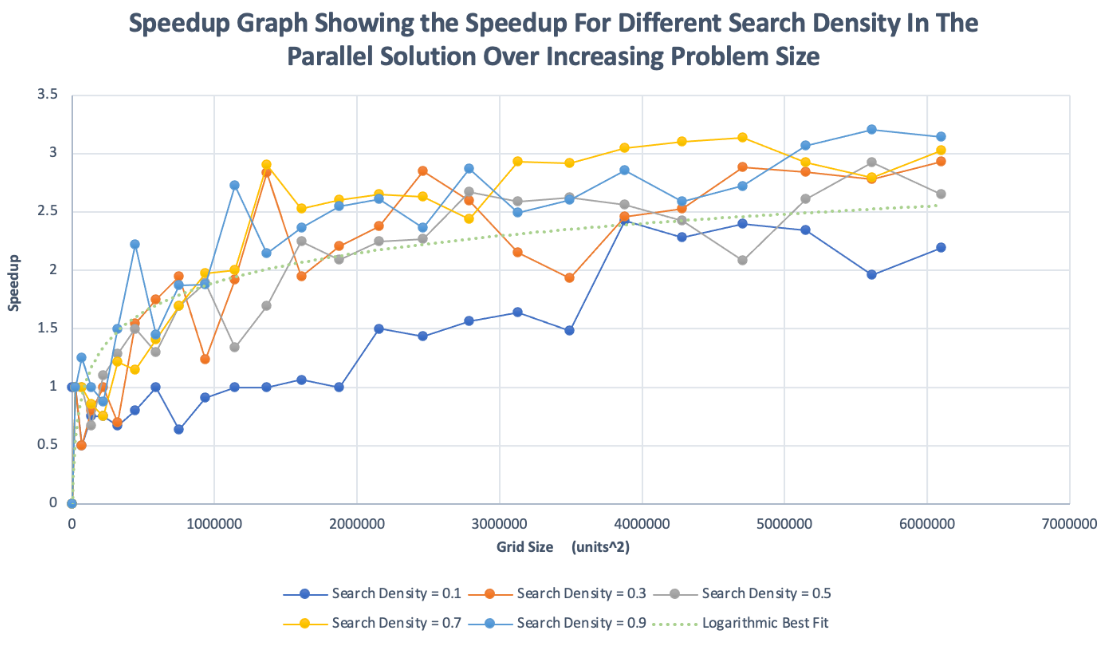
When recording data, the anomalies that occurred were when speed up would be surprisingly low or high and wouldn't follow the trend. The reason for this stems from the random points and the luck of either algorithm to find the one's which lead to the minimum faster. This is shown in the grid points visited and evaluated number.

```
For the same run parameters

Serial solution:
Time: 9 ms                           vs Time: 11 ms
Grid points visited: 33110  (66%)     Grid points visited: 33027  (66%)
Grid points evaluated: 48471  (97%)   Grid points evaluated: 48499  (97%)

Parallel solution:
Time: 12 ms                          vs Time: 14 ms
Grid points visited: 33010  (66%)    Grid points visited: 33125  (66%)
Grid points evaluated: 48437  (97%)  Grid points evaluated: 48459  (97%)

Both calculated the Global minimum = -44735 at x=-26,4 y=6,4
```

In this way tests could vary from the average expected time due to this 'luck of the draw'. Where a speed up of about 0.9 can be achieved through pairing the 11ms and the 12ms, a speed up of 0.6 would be anomalous, but not incorrect, through pairing the 9ms to 14ms.

Speedup Graphs for Search Density

For MacOS:

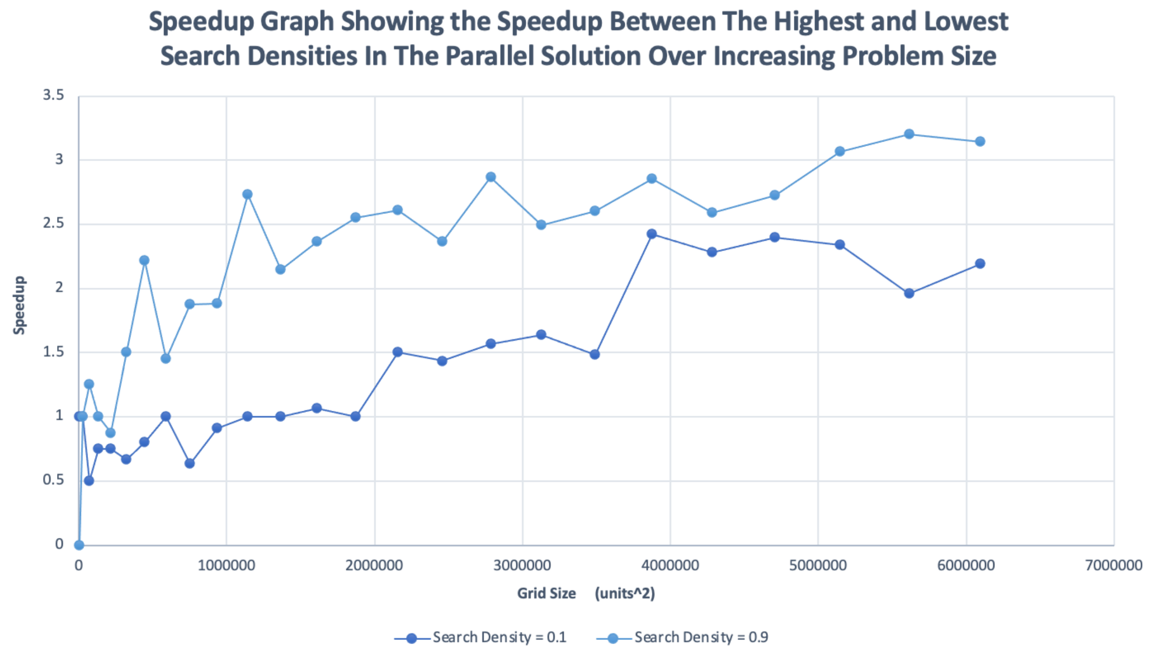


Caption: The above graph was processed to exclude outliers and a logarithmic best fit is included to show the relationship. A best fit line is included to study shape and trend.

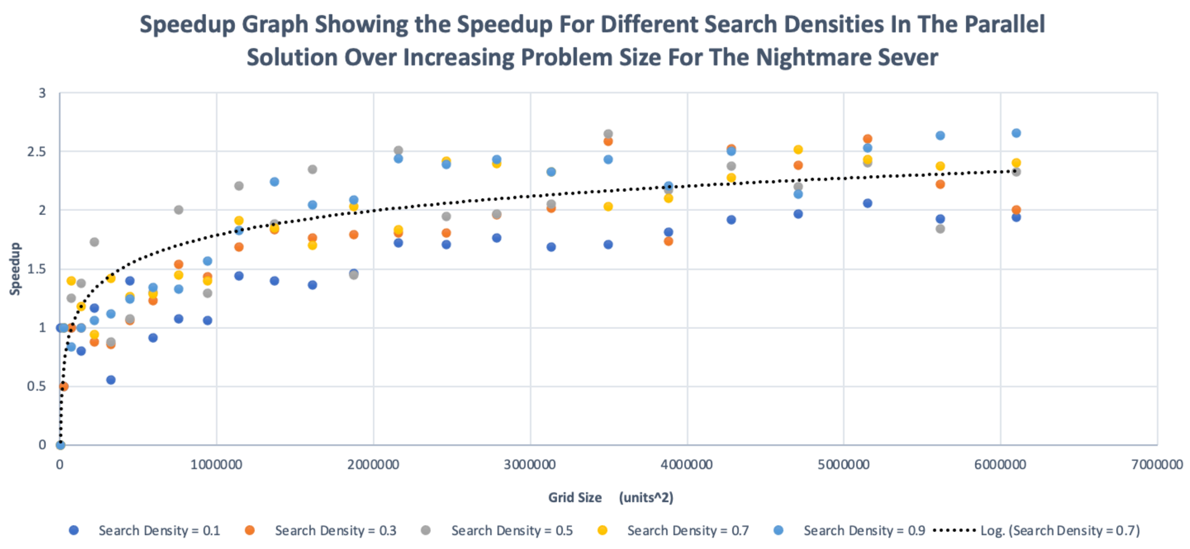
Over all the search densities/along the best fit, the algorithm is shown to be scalable as speedup does increase linearly until a grid size of about 1 000 000, where it reaches its optimal performance. After that the linear relationship deteriorates but still continues to increase to a maximum speedup of 3.2. This is a realistic value since having 4 cores would mean an ideal(maximum) speedup of 4. The algorithm could be short on speedup due to existing serial code, but it is more likely that the operating system had other background programs running, therefore 3.2 is definitely close to what I expected.

The parallel algorithm performs well at grid sizes greater than 1 400 000 because speedup is consistently higher (about double) for these larger grid sizes. This excludes speedup for search density = 0.1, where the parallel program clearly didn't perform as well until the grid size was much larger. Therefore, it is not grid size alone that maximizes speedup (though it is a main contributor).

As shown in previous sections, the number of searches is calculated using grid size and search density. Therefore, to show how the parallel algorithm scales with number of searches, the effect of increasing search density is also applicable and completes the picture. Using the formula, the increase of search density from 0.1 to 0.9 increases the number of searches by 900%. This is plotted below for MacOS.



Therefore, search density also plays a part in increasing speed up because a search density of 0.9 always yielded higher speedup than 0.1 across all grid sizes. Therefore, it is a higher number of total searches which leads to the best performance.

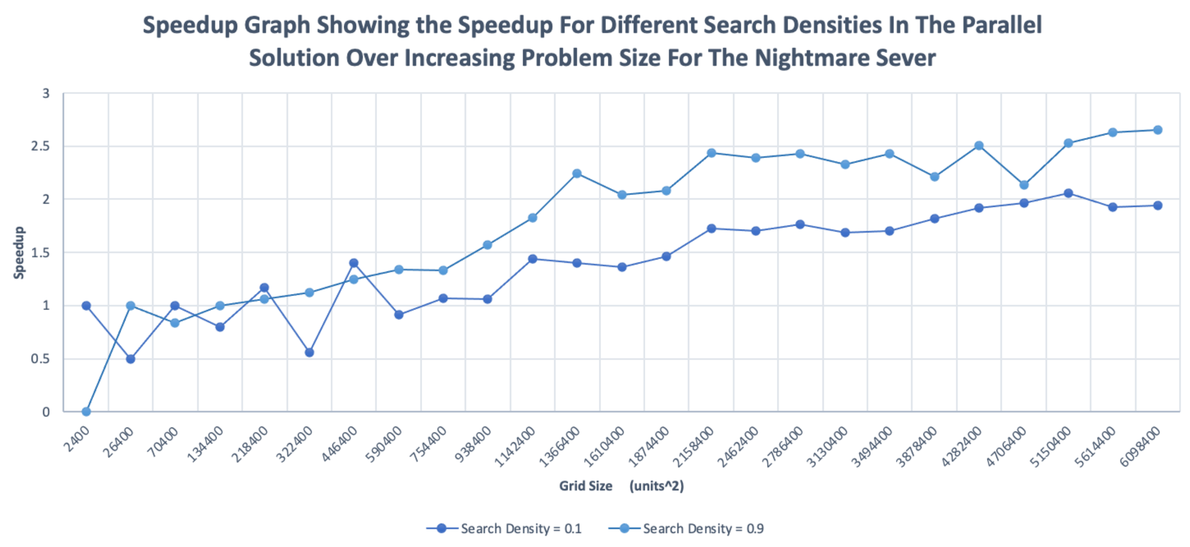


The same trend is found for the program when run on the nightmare server. Since there are so many more available cores, the increase in speedup over the serial solution skyrockets with a much steeper gradient as shown. It therefore plateaus much sooner as expected.

The point at which speedup doubles occurs at a larger grid size than MacOS at around 2 000 000 units^2. This means that the parallel program performs best on nightmare for very large problem sizes. This is what can be expected since there is more processing power available.

These measurements are less reliable than my PC because Nightmare has significantly more traffic which varies unpredictably depending on who is running their programs on the server.

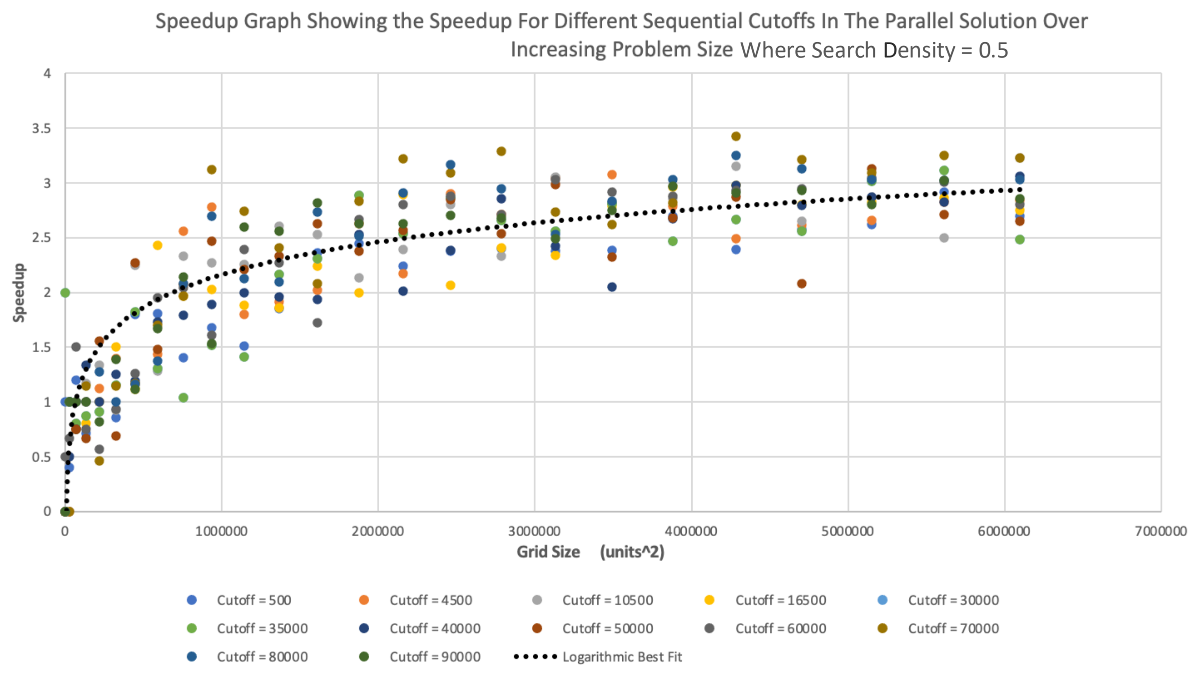
In this way, the maximum speedup of 2.65 is no way near the ideal expected of 8 cores. This means that there was someone else using the cores when my program was running, and I didn't get to use all 8 at all times. This is supported by the below graph:



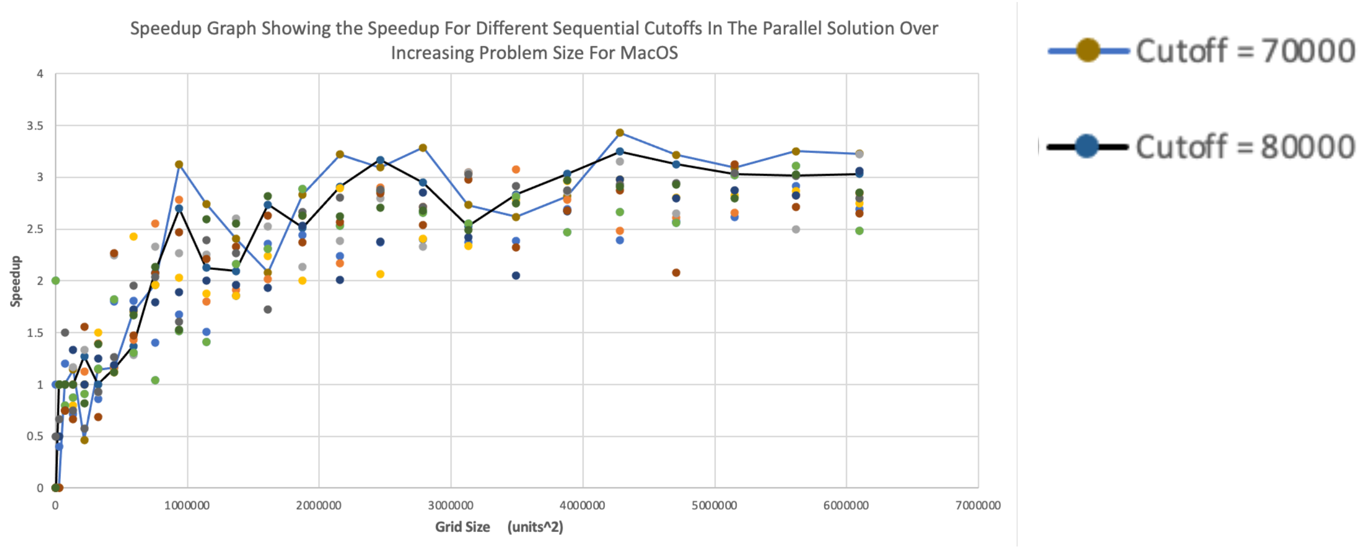
On the MacOS architecture the same parameters resulted in a greater difference for the 900% problem size increase. What likely occurred here is that the unpredictable management of threads by the OS of nightmare allocated the threads in an uneven way for each of the trials according to which cores were available. Someone else's program essentially 'took' the cores and left my threads waiting in some configuration. My PC is much more static and so predictable results were observed.

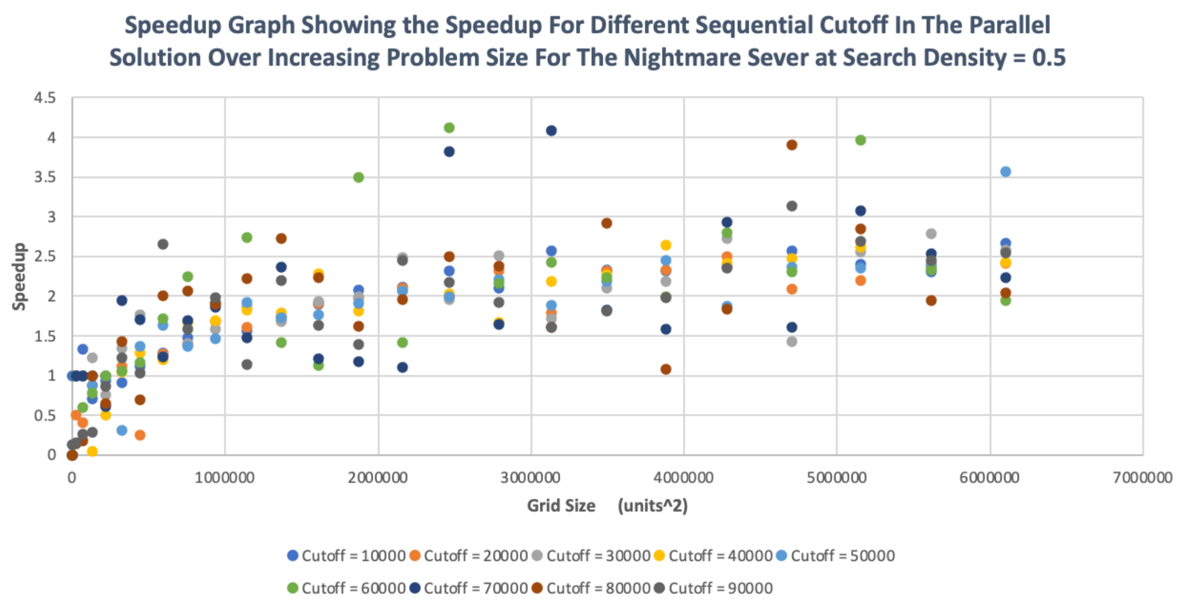
Speedup Graphs for Sequential Cut-off

The sequential cut-off represents the 'serial parts' of the parallel program because it is how many searches a single thread will do. This is plotted below for MacOS:



It may be intuitive that the higher sequential cut-offs lower speedup due to the seriality but, again, problem size is proven to be the main contributor to speedup. The various cut-offs affected speedup inconsistently if barely at all compared to the identical logarithmic increase of speedup over increasing grid sizes.
The higher sequential cut-offs did, however, yield the highest speed up of 3.2 once again, thus validating the previous data. This is shown more clearly below:





In the volatility of nightmare, the unpredictable nature of thread shines in the above graph where different sequential cut-offs try to have the logarithmic trend. This test was run after the previous one and already much higher speed ups are being achieved. The behaviour of threads in this environment causes more outliers than on MacOS. The logarithmic trend is still noticeable although, again, not as reliable.

Conclusions:

Is parallelization worth it for this problem?

While empirically the algorithm is short 0.8 speedup, in practice it is very difficult to get ideal speedup as it is. Getting so close definitely means that the parallelization is worth it, but in addition it is sustained to conclude that it is only worth it for a very large problem size since this is where maximum speed up occurred for both architectures. The difference of 5ms for small problem sizes fails in comparison to 200ms for larger problem sizes.

The ideal implementation of this algorithm would be by a user who has no idea what a certain complex function looks like. In this case, a high number of searches would be needed anyway in order to find the correct global minimum. Therefore, using parallelization is worth it to tackle this problem, since the problem size lies in the range where the parallel algorithm has its best performance. If it were being applied on a 1D function I would not recommend it.