# CSC2002S Report

## Methods

My parallelization approach was to swap the serial solution given to us to a parallel solution by using the Fork / Join Framework where each thread will be given a certain number of searches that it will complete. The amounts given to each thread is determined by the Sequential cutoff of the program which can be changed accordingly. Each thread would perform their series of searches finding the smallest point in their array of searches which then after all threads are finished running the smallest point of all the threads is taken from them, giving us the minimum of the grid points. A major issue I had was deciding on whether to use the RecursiveAction or the RecursiveTask class as one class returns a value whereas the other doesn't. In the end I used the RecursiveTask class as we are performing a series of searches and not returning a value each time. I optimized the thread searches by only changing the minimum each time if it is lower than the other thread searches giving less run time and more performance.

I validated my algorithms by using the Rosenbrock Function which is used to test the performance of optimization algorithms such as the ones I am using. To use the function I ran the algorithm twice, once where I used the algorithm that I used to find the global minimum and the second time using the Rosenbrock function in place of the function I used before and seeing if the global minimum is equal to the global minimum of the Rosenbrock function which is 0, which it is. This proves the validity of my algorithm in finding the global minimum to be very correct and valid.

Parallel (arguments = 1000 1000 1 50 1 50 0.1)

My Function

```
Time: 56 ms
Grid points visited: 338938  (34%)
Grid points evaluated: 339518  (34%)
Global minimum: -67313 at x=11,0 y=6,3
```

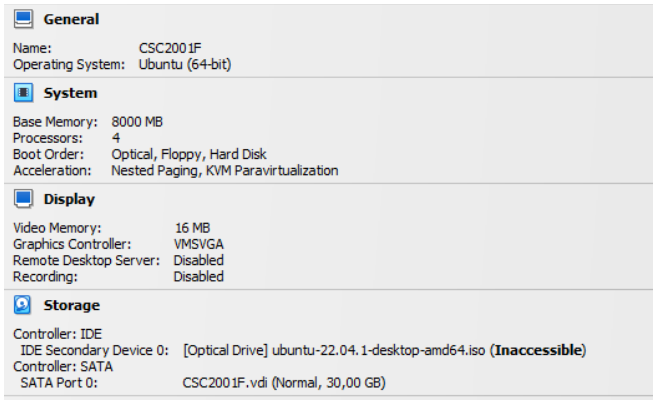
Rosenbrock Function

```
Time: 41 ms
Grid points visited: 123545  (12%)
Grid points evaluated: 204258  (20%)
Global minimum: 0 at x=2,9 y=8,5
```

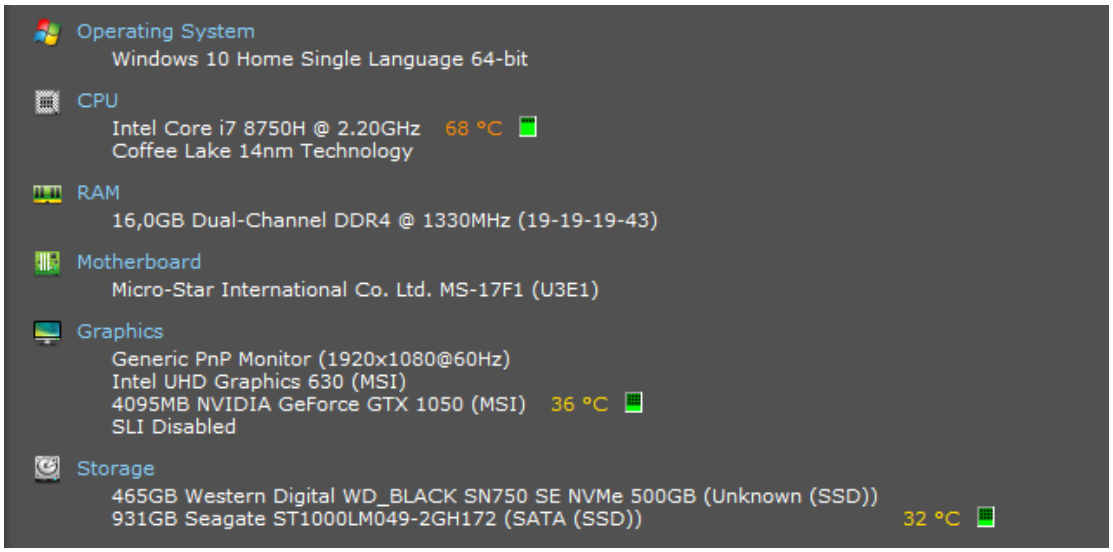
I benchmarked my algorithm by creating two tables for two different machines with different specifications and running the serial version and the parallel version of the program. I then compared the run time taken by the serial and parallel version of the program while varying the grid points, search density and with the parallel version the sequential cutoff and creating a speedup that occurred between going from the serial version to the parallel version. To get more accurate and optimized results I took the average of 5 runs of the program and inputted that into my table rather than taking 1 single run as it may be possibly flawed due to external factors.

I benchmarked my results on two different machine architectures both containing 4 cores but different specifications with one being run on a Java Virtual Machine (JVM) and the other being run on my laptop with windows 10 with more higher end hardware. The JVM had to for some grid sizes of 10000 x 10000 had to have more memory allocated to it to avoid Java Heap Space errors.

Java Virtual Machine



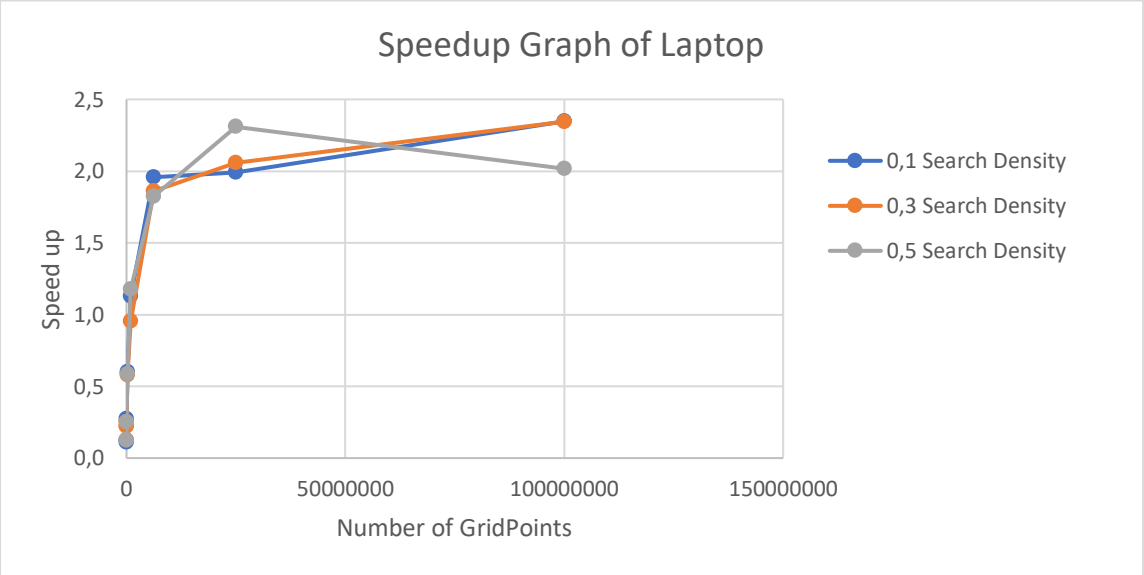

My laptop

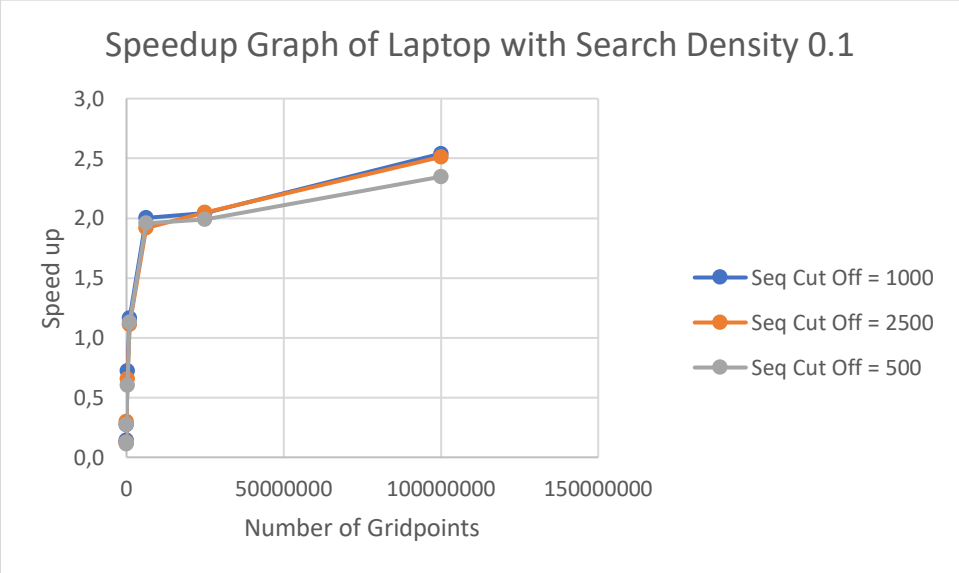



The only major problems or difficulties I encountered was the varying results of the run time of the programs as some results would be so spontaneous and others be very similar to the other results. In other words, the time taken to perform the program would continue to be similar to other runs then all of a sudden a run time much different to the other results would occur. Also the running out of memory on my Java Virtual Machine for big grid sizes for the serial solution of the algorithm which didn't occur when I ran it on my parallel version.

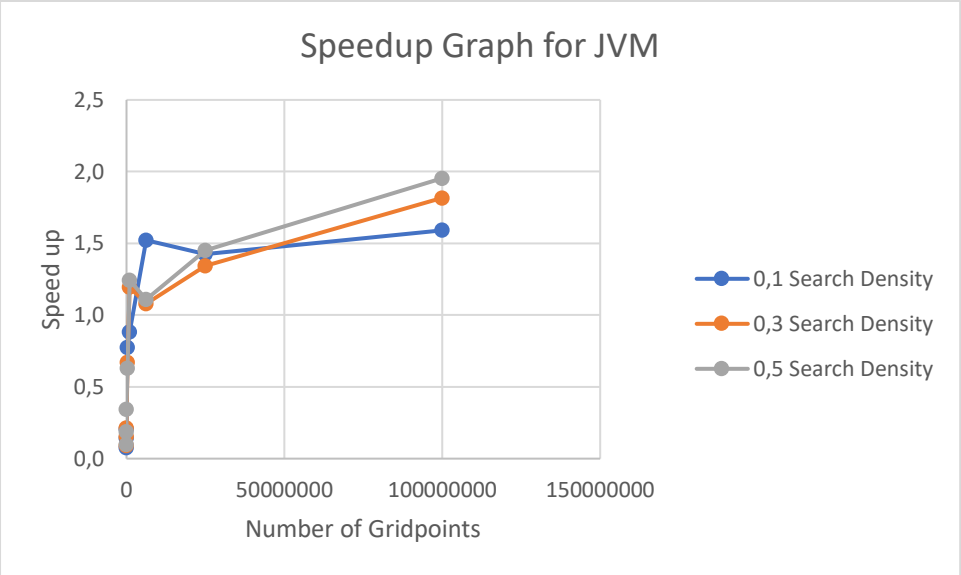
## Results

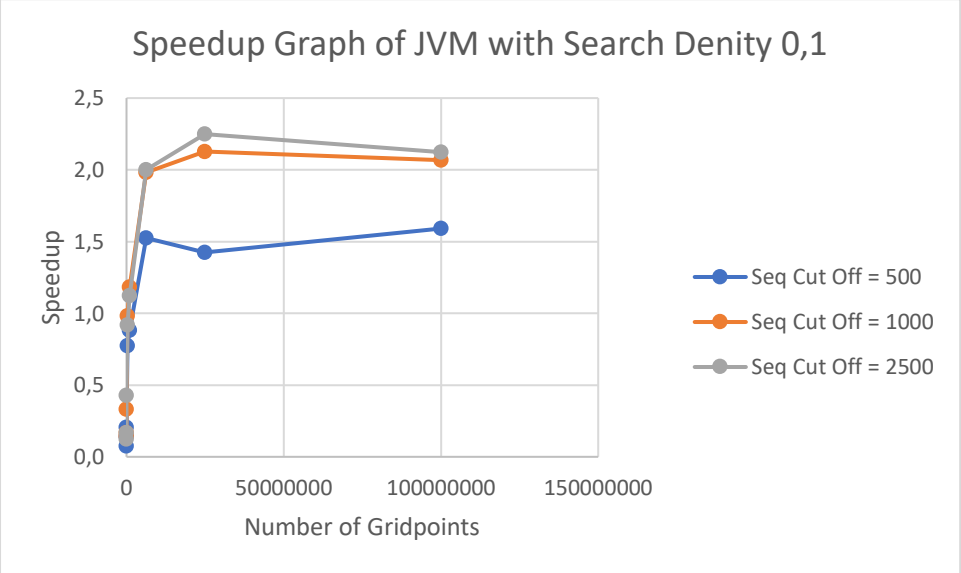
Laptop Results (refer to Results A)



JVM Results (refer to Results B)




## Discussion

My results give the impression that the grid sizes that it performs the best for is the larger grid sizes such as the 5000 x 5000 and 10000 x 10000. As with smaller grid sizes the program is performing slower than the serial version. I believe the reason for this is that with the Fork / Join framework or divide and conquer algorithms, the maximum speedup of n / log(n) grows exponentially with the increase in problem size which in this case is the number of gridpoints. Therefore as the gridpoints are increased significantly the speedup will also increase at a very high exponential rate. The maximum speedup I achieved with 2.5 times, which in comparison to my the max speedup of this program which is 4, due to both machines having 4 cores, it is quite close to the max speedup but if I was to reduce the percentage of sequential algorithms in my program I could probably achieve a speedup closer to that of 4 according to Ahmdal's Law. My measurements can not be as reliable as I

would want them to be as many factors affect the runtime of my program as with multithreading the performance of the program relies very heavily on the hardware specifications of the machine it is running from which can be seen in the differing results from my laptop and my JVM. The JVM is being run by my laptop adding a extra layer of performance creating the worse speedups and runtimes. To make my measurements as accurate as possible I ran each argument of the program 5 times and took 3 of the results that were in a good range of each other and plotted the average of these 3 results in my table and used that measurement when producing my results making the results more reliable and reducing the effect of the machine performance. There are some anomalies as the code does include a race condition, as two threads can visit the same point at the same time and mark it as visited, when in the idea situation the threads should recognize that a thread is visiting that point and try and find work elsewhere. This race condition creates these anomalies of different minimums in my program but for this assignment we have chosen to ignore it. Other than the race condition I did achieve some spikes in my graphs occurring from bad performance from the machine side as with most software when used with parallel programming, its required to warm up before producing results as the results may be flawed if its not “warm”.

## Conclusion

To draw a conclusion from this assignment, I would say that using parallelization to go at this problem in Java is a very good idea and worth as long as you are working with larger values of grid points as then then your results will be produced a lot quicker than a serial version of the program. So if you are working with smaller grid sizes using parallelization will only slow down the production of results due to the speedup being less than 1. Therefore I’d recommend only parallelizing this program if the grid point sizes you are trying to find the minimum of are greater than **1000000** or more as then you will achieve a much greater speedup that will only increase exponentially with larger grid points.

## Appendix

### Laptop Results

| Rows | Coloumns | No. of GridPoints | xMin | xMax | yMin | yMax | Search Density | Time (ms) | Sequential Cutoff N/A | Serial Time (ms) | Speed up |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 5 | **25** | 1 | 50 | 1 | 50 | 0,1 | 8 | 500 | 1 | **0,1** |
| 10 | 10 | **100** | 1 | 50 | 1 | 50 | 0.1 | 9 | 500 | 1 | **0,1** |
| 100 | 100 | **10000** | 1 | 50 | 1 | 50 | 0.1 | 11 | 500 | 3 | **0,3** |
| 500 | 500 | **250000** | 1 | 50 | 1 | 50 | 0,1 | 35 | 500 | 21 | **0,6** |
| 1000 | 1000 | **1000000** | 1 | 50 | 1 | 50 | 0,1 | 63 | 500 | 71 | **1,1** |
| 2500 | 2500 | **6250000** | 1 | 50 | 1 | 50 | 0,1 | 305 | 500 | 597 | **2,0** |
| 5000 | 5000 | **25000000** | 1 | 50 | 1 | 50 | 0,1 | 1320 | 500 | 2628 | **2,0** |
| 10000 | 10000 | **100000000** | 1 | 50 | 1 | 50 | 0,1 | 5862 | 500 | 13758 | **2,3** |
| 5 | 5 | **25** | 1 | 50 | 1 | 50 | 0,3 | 9 | 500 | 2 | **0,2** |
| 10 | 10 | **100** | 1 | 50 | 1 | 50 | 0,3 | 9 | 500 | 2 | **0,2** |
| 100 | 100 | **10000** | 1 | 50 | 1 | 50 | 0,3 | 12 | 500 | 3 | **0,3** |
| 500 | 500 | **250000** | 1 | 50 | 1 | 50 | 0,3 | 38 | 500 | 22 | **0,6** |
| 1000 | 1000 | **1000000** | 1 | 50 | 1 | 50 | 0,3 | 85 | 500 | 81 | **1,0** |
| 2500 | 2500 | **6250000** | 1 | 50 | 1 | 50 | 0,3 | 360 | 500 | 670 | **1,9** |
| 5000 | 5000 | **25000000** | 1 | 50 | 1 | 50 | 0,3 | 1388 | 500 | 2858 | **2,1** |
| 10000 | 10000 | **100000000** | 1 | 50 | 1 | 50 | 0,3 | 5639 | 500 | 13224 | **2,3** |
| 5 | 5 | **25** | 1 | 50 | 1 | 50 | 0,5 | 8 | 500 | 1 | **0,1** |
| 10 | 10 | **100** | 1 | 50 | 1 | 50 | 0,5 | 8 | 500 | 1 | **0,1** |
| 100 | 100 | **10000** | 1 | 50 | 1 | 50 | 0,5 | 12 | 500 | 3 | **0,3** |
| 500 | 500 | **250000** | 1 | 50 | 1 | 50 | 0,5 | 41 | 500 | 24 | **0,6** |
| 1000 | 1000 | **1000000** | 1 | 50 | 1 | 50 | 0,5 | 80 | 500 | 94 | **1,2** |
| 2500 | 2500 | **6250000** | 1 | 50 | 1 | 50 | 0,5 | 390 | 500 | 711 | **1,8** |
| 5000 | 5000 | **25000000** | 1 | 50 | 1 | 50 | 0,5 | 1325 | 500 | 3059 | **2,3** |
| 10000 | 10000 | **100000000** | 1 | 50 | 1 | 50 | 0,5 | 6753 | 500 | 13620 | **2,0** |
| 5 | 5 | **25** | 1 | 50 | 1 | 50 | 0,1 | 7 | 1000 | 1 | **0,1** |
| 10 | 10 | **100** | 1 | 50 | 1 | 50 | 0.1 | 8 | 1000 | 1 | **0,1** |
| 100 | 100 | **10000** | 1 | 50 | 1 | 50 | 0.1 | 11 | 1000 | 3 | **0,3** |
| 500 | 500 | **250000** | 1 | 50 | 1 | 50 | 0,1 | 29 | 1000 | 21 | **0,7** |
| 1000 | 1000 | **1000000** | 1 | 50 | 1 | 50 | 0,1 | 61 | 1000 | 71 | **1,2** |
| 2500 | 2500 | **6250000** | 1 | 50 | 1 | 50 | 0,1 | 298 | 1000 | 597 | **2,0** |
| 5000 | 5000 | **25000000** | 1 | 50 | 1 | 50 | 0,1 | 1287 | 1000 | 2628 | **2,0** |
| 10000 | 10000 | **100000000** | 1 | 50 | 1 | 50 | 0,1 | 5417 | 1000 | 13758 | **2,5** |
| 5 | 5 | **25** | 1 | 50 | 1 | 50 | 0,1 | 8 | 2500 | 1 | **0,1** |
| 10 | 10 | **100** | 1 | 50 | 1 | 50 | 0,1 | 8 | 2500 | 1 | **0,1** |
| 100 | 100 | **10000** | 1 | 50 | 1 | 50 | 0,1 | 10 | 2500 | 3 | **0,3** |
| 500 | 500 | **250000** | 1 | 50 | 1 | 50 | 0,1 | 32 | 2500 | 21 | **0,7** |
| 1000 | 1000 | **1000000** | 1 | 50 | 1 | 50 | 0,1 | 64 | 2500 | 71 | **1,1** |
| 2500 | 2500 | **6250000** | 1 | 50 | 1 | 50 | 0,1 | 311 | 2500 | 597 | **1,9** |
| 5000 | 5000 | **25000000** | 1 | 50 | 1 | 50 | 0,1 | 1284 | 2500 | 2628 | **2,0** |
| 10000 | 10000 | **100000000** | 1 | 50 | 1 | 50 | 0,1 | 5476 | 2500 | 13758 | **2,5** |

S

JVM Results

| Rows | Coloumns | No. of GridPoints | xMin | xMax | yMin | yMax | Search Density | Time (ms) | Sequential Cutoff N/A | Serial Time (ms) | Speed up |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 5 | **25** | 1 | 50 | 1 | 50 | **0,1** | 13 | 500 | 1 | 0,1 |
| 10 | 10 | **100** | 1 | 50 | 1 | 50 | 0.1 | 7 | 500 | 1 | 0,1 |
| 100 | 100 | **10000** | 1 | 50 | 1 | 50 | 0.1 | 29 | 500 | 6 | 0,2 |
| 500 | 500 | **250000** | 1 | 50 | 1 | 50 | 0,1 | 75 | 500 | 58 | 0,8 |
| 1000 | 1000 | **1000000** | 1 | 50 | 1 | 50 | 0,1 | 177 | 500 | 156 | 0,9 |
| 2500 | 2500 | **6250000** | 1 | 50 | 1 | 50 | 0,1 | 673 | 500 | 1025 | 1,5 |
| 5000 | 5000 | **25000000** | 1 | 50 | 1 | 50 | 0,1 | 3081 | 500 | 4389 | 1,4 |
| 10000 | 10000 | **100000000** | 1 | 50 | 1 | 50 | 0,1 | 10693 | 500 | 17012 | 1,6 |
| 5 | 5 | **25** | 1 | 50 | 1 | 50 | **0,3** | 11 | 500 | 1 | 0,1 |
| 10 | 10 | **100** | 1 | 50 | 1 | 50 | 0,3 | 13 | 500 | 2 | 0,2 |
| 100 | 100 | **10000** | 1 | 50 | 1 | 50 | 0,3 | 28 | 500 | 6 | 0,2 |
| 500 | 500 | **250000** | 1 | 50 | 1 | 50 | 0,3 | 91 | 500 | 61 | 0,7 |
| 1000 | 1000 | **1000000** | 1 | 50 | 1 | 50 | 0,3 | 153 | 500 | 183 | 1,2 |
| 2500 | 2500 | **6250000** | 1 | 50 | 1 | 50 | 0,3 | 983 | 500 | 1060 | 1,1 |
| 5000 | 5000 | **25000000** | 1 | 50 | 1 | 50 | 0,3 | 3137 | 500 | 4218 | 1,3 |
| 10000 | 10000 | **100000000** | 1 | 50 | 1 | 50 | 0,3 | 9398 | 500 | 17079 | 1,8 |
| 5 | 5 | **25** | 1 | 50 | 1 | 50 | **0,5** | 10 | 500 | 1 | 0,1 |
| 10 | 10 | **100** | 1 | 50 | 1 | 50 | 0,5 | 16 | 500 | 3 | 0,2 |
| 100 | 100 | **10000** | 1 | 50 | 1 | 50 | 0,5 | 29 | 500 | 10 | 0,3 |
| 500 | 500 | **250000** | 1 | 50 | 1 | 50 | 0,5 | 95 | 500 | 60 | 0,6 |
| 1000 | 1000 | **1000000** | 1 | 50 | 1 | 50 | 0,5 | 149 | 500 | 185 | 1,2 |
| 2500 | 2500 | **6250000** | 1 | 50 | 1 | 50 | 0,5 | 994 | 500 | 1101 | 1,1 |
| 5000 | 5000 | **25000000** | 1 | 50 | 1 | 50 | 0,5 | 3172 | 500 | 4598 | 1,4 |
| 10000 | 10000 | **100000000** | 1 | 50 | 1 | 50 | 0,5 | 9740 | 500 | 19034 | 2,0 |
| 5 | 5 | **25** | 1 | 50 | 1 | 50 | 0,1 | 6 | 1000 | 1 | 0,2 |
| 10 | 10 | **100** | 1 | 50 | 1 | 50 | 0.1 | 7 | 1000 | 1 | 0,1 |
| 100 | 100 | **10000** | 1 | 50 | 1 | 50 | 0.1 | 18 | 1000 | 6 | 0,3 |
| 500 | 500 | **250000** | 1 | 50 | 1 | 50 | 0,1 | 59 | 1000 | 58 | 1,0 |
| 1000 | 1000 | **1000000** | 1 | 50 | 1 | 50 | 0,1 | 132 | 1000 | 156 | 1,2 |
| 2500 | 2500 | **6250000** | 1 | 50 | 1 | 50 | 0,1 | 517 | 1000 | 1025 | 2,0 |
| 5000 | 5000 | **25000000** | 1 | 50 | 1 | 50 | 0,1 | 5442 | 1000 | 11579 | 2,1 |
| 10000 | 10000 | **100000000** | 1 | 50 | 1 | 50 | 0,1 | 8226 | 1000 | 17012 | 2,1 |
| 5 | 5 | **25** | 1 | 50 | 1 | 50 | 0,1 | 6 | 2500 | 1 | 0,2 |
| 10 | 10 | **100** | 1 | 50 | 1 | 50 | 0,1 | 8 | 2500 | 1 | 0,1 |
| 100 | 100 | **10000** | 1 | 50 | 1 | 50 | 0,1 | 14 | 2500 | 6 | 0,4 |
| 500 | 500 | **250000** | 1 | 50 | 1 | 50 | 0,1 | 63 | 2500 | 58 | 0,9 |
| 1000 | 1000 | **1000000** | 1 | 50 | 1 | 50 | 0,1 | 139 | 2500 | 156 | 1,1 |
| 2500 | 2500 | **6250000** | 1 | 50 | 1 | 50 | 0,1 | 512 | 2500 | 1025 | 2,0 |
| 5000 | 5000 | **25000000** | 1 | 50 | 1 | 50 | 0,1 | 5148 | 2500 | 11579 | 2,2 |
| 10000 | 10000 | **100000000** | 1 | 50 | 1 | 50 | 0,1 | 8012 | 2500 | 17012 | 2,1 |