**Hunt for the dungeon master – PCP 1**

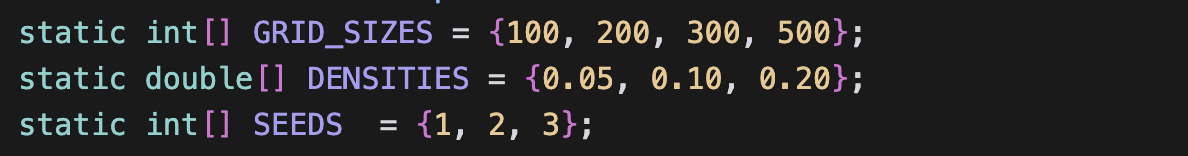
In this PCP assignment I implemented a parallel algorithm to locate the greatest mana value in the SoloLevelling 2D grid. This implementation involved the monte carlo method along witht the fork/join framework using join() for synchronization to create multiple threads.

I used claude (generative AI model) along with my own knowledge to create 2 java scrips, serial and parallel profilers that ran multiple different tests of each version consisting of grid size (20, 50, 100, 300, 500) densities of (0.05, 0.10, 0.20, 0.30, 0.50, 0.60, 0.70) and a random seed of 42 for reproducibility. These profilers ran each test 3 times and took the average execution time of each one and recorded the results in a txt file.

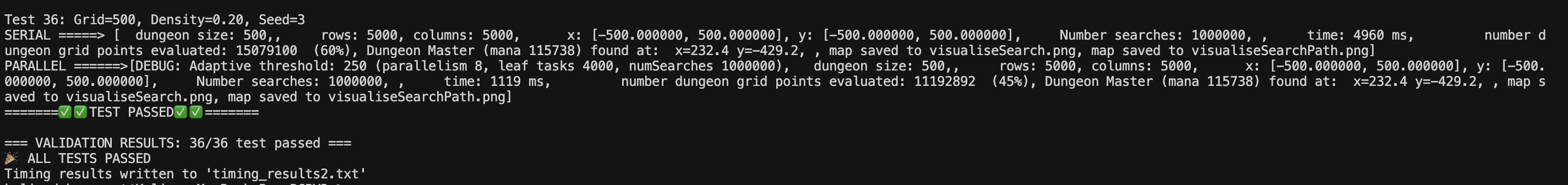
Again I used claude to generate me a python script that took these txt files with the averaged times for each test to generate speedup graphs vs grid size and density

**Methods**

**Validation of correctness**

After implementing the parallel algorithm, I created a ValidationScript.java file that ran both versions simultaneously while parsing the timed outputs and comparing them and check if the values for x, y and mana matches with a margin for error of +-0.1 for the x & y values. I used and looped through these values for testing:

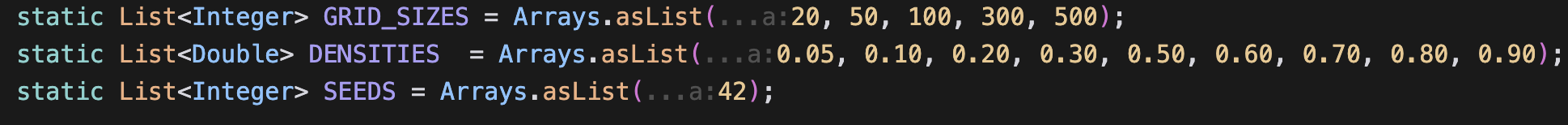
After running through every combination of test my program returned 100% accuracy and all 36 test cases were passed . the timings were saved in a txt file.



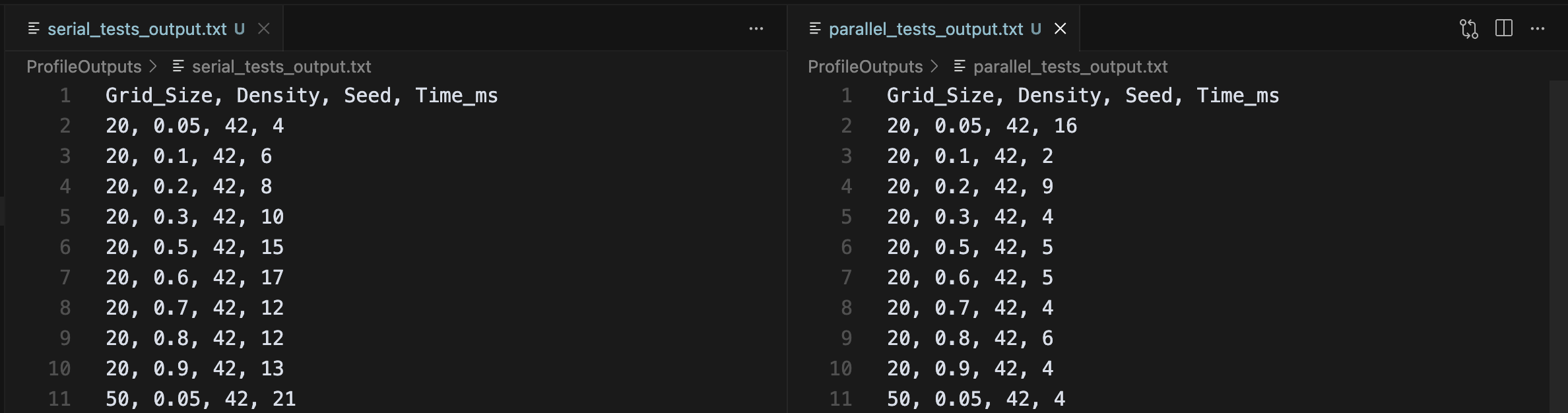
The correctness of this was validated by my program on my local machine (MacBook Pro intel i5). I then ran the same file on the UCT nightmare server which also validated the correctness of my parallel implementation while passing all 36 test cases from which I can conclude that my parallel implementation is correctly implemented when it comes to the functionality of the programs.

**Profiling**

The serial and parallel implementations were profiled via the **SerialProfiler.java** & **ParallelProfiler.java** files respectively. This was done to acquire the execution times for varying inputs, below is the list of parameters I used.



Each test was run 3 times and averaged across the 3 executions for reliability and to prevent system fluctuations from affecting the results. These averaged timings were stored in **serial\_tests\_output.txt** and **parallel\_tests\_outputs.txt** which look something like this:

****

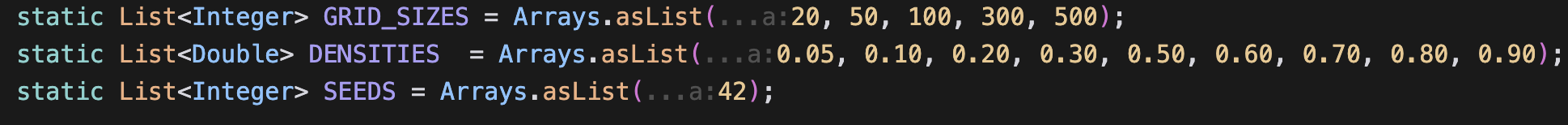
**Benchmarking**

Compare end to end execution time serial vs parallel across a range of different inputs.

My local machine 🡪 Intel(R) Core(TM) i5-8257U CPU @ 1.40GHz, 🡪 8 core.

Nightmare server 🡪 Linux 5.15, Intel Xeon E5620 @ 2.40GHz (8 cores), 47 GiB RAM, Java 17.0.6.

Tests used – Same as the profiling parameters since I am using the values stored from the profilers:



The parallel implementation uses a fork/join threshold(cutoff) that is computed dynamically at runtime based on the workload and parallelism available. This is to achieve the best possible results for each sample size. For small tests, smaller thresholds, larger will calculate a larger threshold to use. I capped the threshold at 2056 to prevent it from becoming too large.

I benchmarked both my versions on my local machine by running **make profile-both** which runs both profilers and captures the timings. The threshold is calculated dynamically to improve performance and efficiency and further testing revealed that a threshold of 2056 is optimal on average for all test cases run.

Speedup graphs were then generated by joining the two CSV files produced by the profilers on (Grid\_size, Density, Seed) and dividing (serial\_time\_ms / parallel\_time\_ms)

**Optimum search density**

On my machine after running the **ParallelProfiler.java** I used the **GRAPH.py** file to generate speedup graphs and a speedup report and analysis for the serial vs parallel execution times.

On my local machine 🡪 Maximum speed up of **4.85x** was found at - **Grid size: 500, density 0.10**

On the nightmare server 🡪 Maximum speedup of **5.66x** was found at – Grid size 100, Density 0.90

A seed of 42 was used for both for reproducibility.

**Results**

**Validation –** After running the **ValidationScript.java** file on my local machine and the nightmare serve to compare the outputs of the serial vs parallel implementation. I found that on both machines my validation script showed that the parallel implementation is fully correct and returns the exact same values as the serial and all values match. Here are the results, by examining the results we can also conclude the parallel algorithm is quicker than the serial algorithm:

Local machine : results saved to ParallelValidation.txt

A computer screen with white text

AI-generated content may be incorrect.

Nightmare server:

A computer screen with white text

AI-generated content may be incorrect.

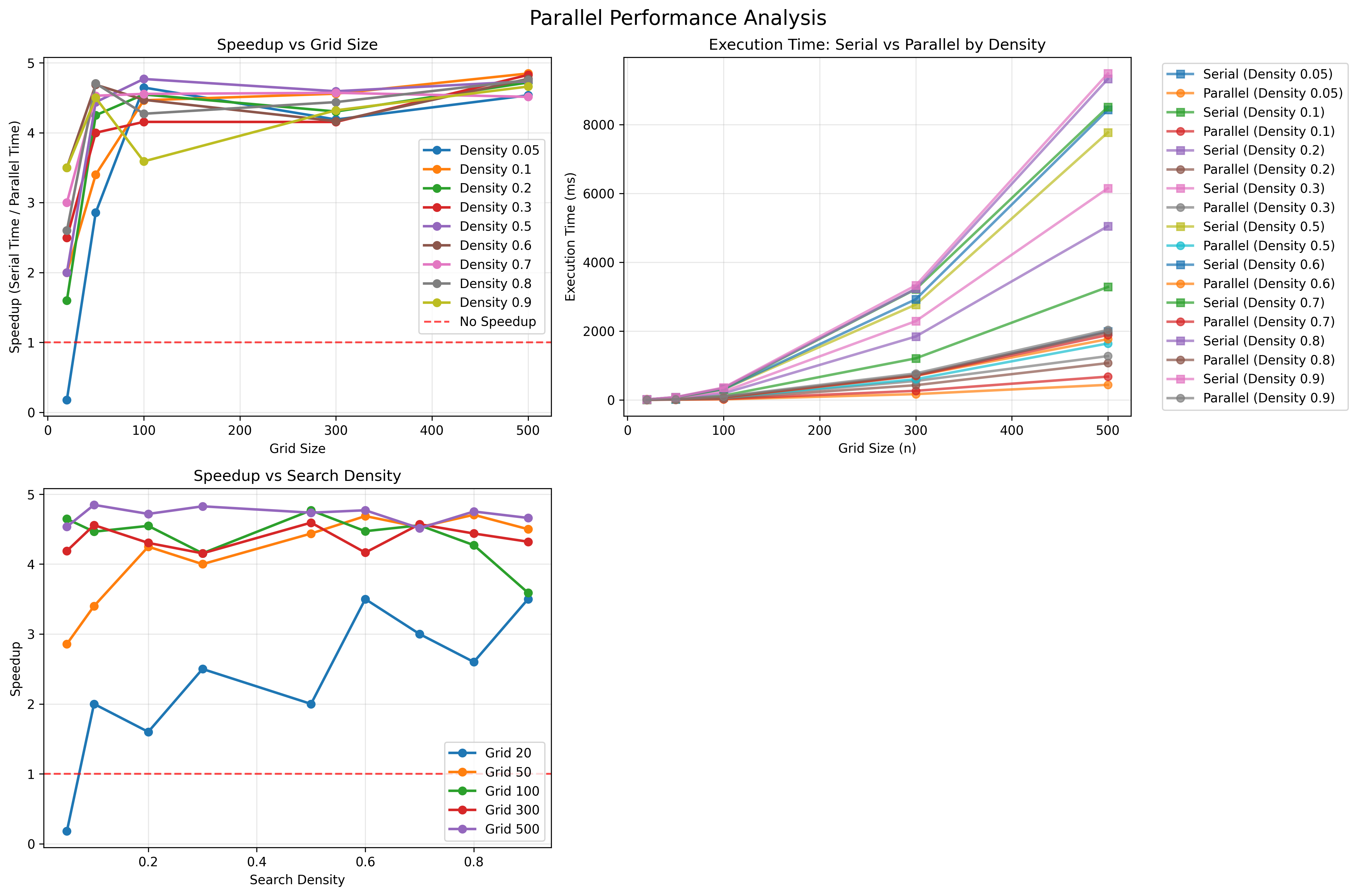
In both versions we can see that the parallel implementation passed all tests, returning the same x, y and mana values as the serial version for the following parameters, proving the correctness of the parallel implementation:

A black background with numbers and symbols

AI-generated content may be incorrect.

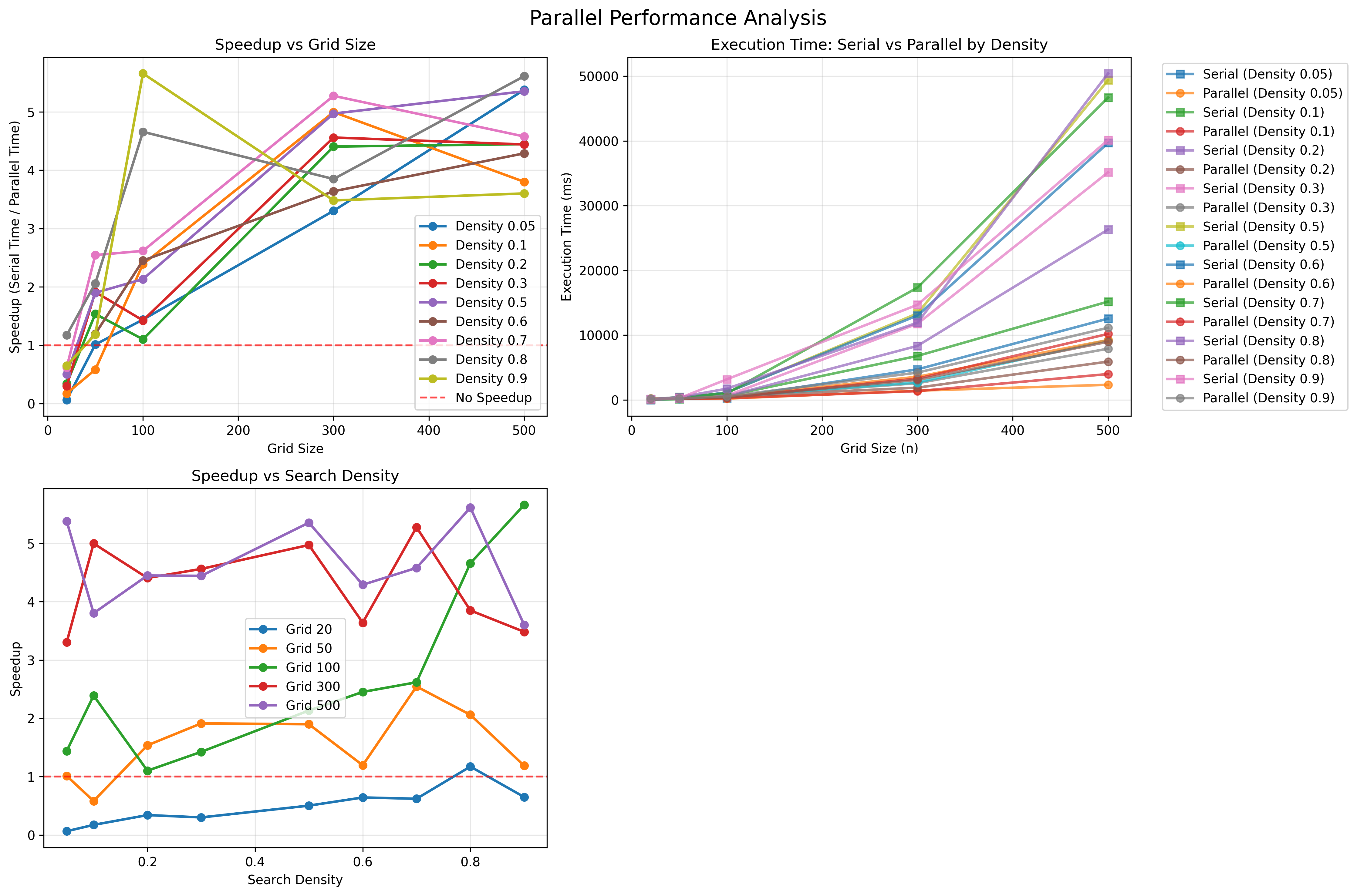
**Benchmarking**

I had claude ai (generative ai model) to create a python Script called GRAPH.py to calculate the speedups based on the results from the parallel and serial profiler outputs for all the tests cases I used and plot the results in a graph. I decided to graph all the different cases to see how the grid size and density affected the speedup in which we found on the local machine:

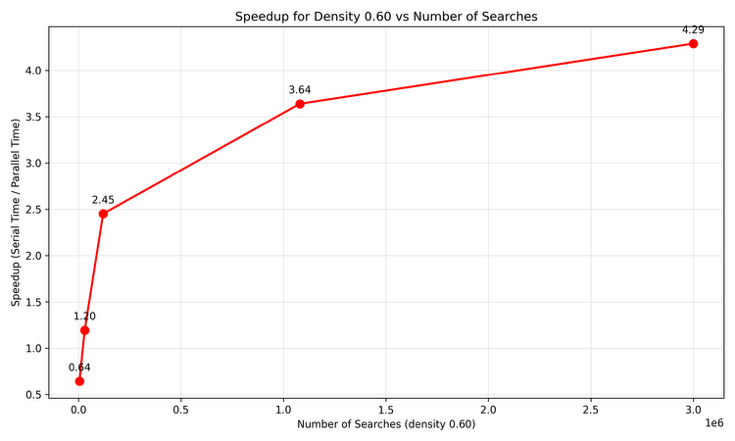


We can see that the maximum speedup achieved was 4.8x at grid size 500.0 and density of 0.10

Nightmare server:

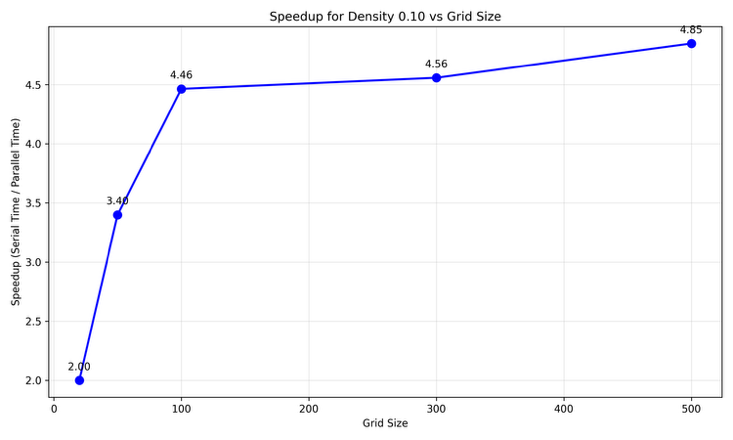
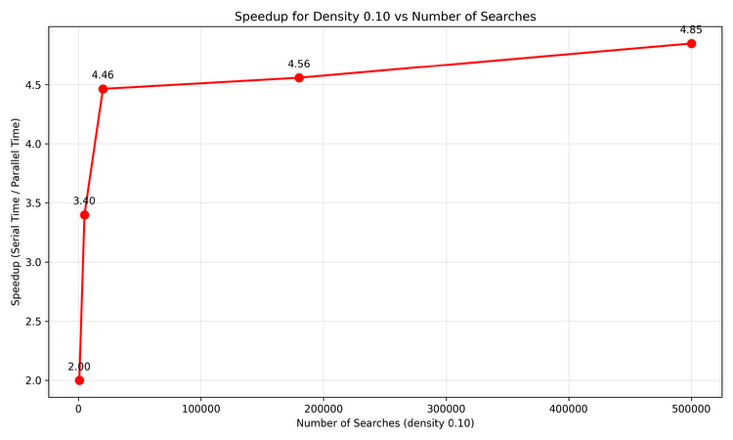


Using the optimal parameters: density 0.10, grid size 500.0 for my local machine and density 0.90 grid size 100.0 for the nightmare server where we found our optimum search density we get

**A graph with a line

AI-generated content may be incorrect.Nightmare server:**

Graph of the optimum search density that achieved the highest speedup on the nightmare server, using the adaptive threshold and capped at 2056. The threshold of 2056 was used most on average.

**Local machine:**

Using the optimum search density for both the server runs and my local machine runs we get the above graphs showing how the parallel algorithm scales vs number of searches and the grid size for multiple tests as well as for the optimum search density.

=== PERFORMANCE ANALYSIS REPORT ===

1. MAXIMUM SPEEDUP ACHIEVED:

Maximum speedup: 4.85x

At grid size: 500.0

At density: 0.1

2. SPEEDUP BY GRID SIZE:

Grid 20: 2.32x average speedup

Grid 50: 4.15x average speedup

Grid 100: 4.39x average speedup

Grid 300: 4.37x average speedup

Grid 500: 4.71x average speedup

3. SPEEDUP BY DENSITY:

Density 0.05: 3.28x average speedup

Density 0.1: 3.85x average speedup

Density 0.2: 3.88x average speedup

Density 0.3: 3.93x average speedup

Density 0.5: 4.11x average speedup

Density 0.6: 4.32x average speedup

Density 0.7: 4.23x average speedup

Density 0.8: 4.15x average speedup

Density 0.9: 4.11x average speedup

4. CASES WHERE PARALLEL IS SLOWER (speedup < 1.0):

Grid 20.0, Density 0.05: 0.18x

=== PERFORMANCE ANALYSIS REPORT ===

1. MAXIMUM SPEEDUP ACHIEVED:

Maximum speedup: 5.66x

At grid size: 100.0

At density: 0.9

2. SPEEDUP BY GRID SIZE:

Grid 20: 0.50x average speedup

Grid 50: 1.55x average speedup

Grid 100: 2.65x average speedup

Grid 300: 4.28x average speedup

Grid 500: 4.61x average speedup

3. SPEEDUP BY DENSITY:

Density 0.05: 2.24x average speedup

Density 0.1: 2.39x average speedup

Density 0.2: 2.37x average speedup

Density 0.3: 2.53x average speedup

Density 0.5: 2.97x average speedup

Density 0.6: 2.44x average speedup

Density 0.7: 3.13x average speedup

Density 0.8: 3.47x average speedup

Density 0.9: 2.92x average speedup

4. CASES WHERE PARALLEL IS SLOWER (speedup < 1.0):

Grid 20.0, Density 0.05: 0.06x

Grid 20.0, Density 0.1: 0.17x

Grid 20.0, Density 0.2: 0.34x

Grid 20.0, Density 0.3: 0.30x

Grid 20.0, Density 0.5: 0.50x

Grid 20.0, Density 0.6: 0.64x

Grid 20.0, Density 0.7: 0.62x

Grid 20.0, Density 0.9: 0.65x

Grid 50.0, Density 0.1: 0.58x