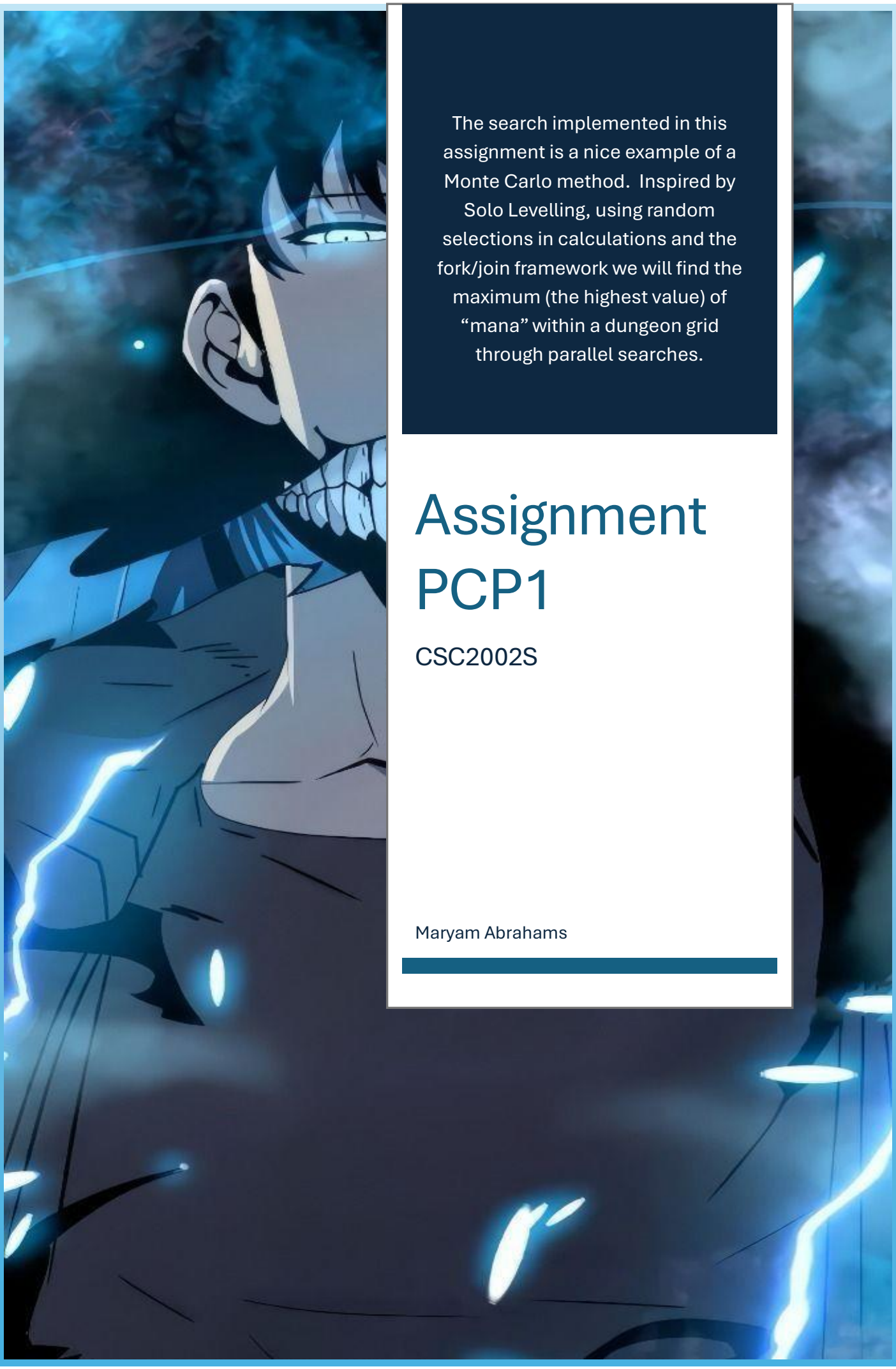


The search implemented in this assignment is a nice example of a Monte Carlo method. Inspired by Solo Levelling, using random selections in calculations and the fork/join framework we will find the maximum (the highest value) of "mana" within a dungeon grid through parallel searches.

Assignment PCP1

CSC2002S

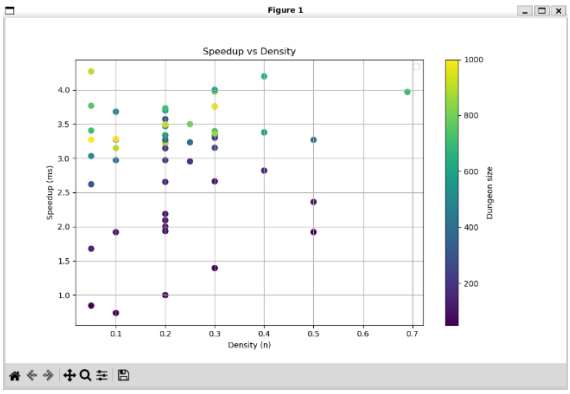
Maryam Abrahams

Optimum Search Density:

To find an optimum search density for my speedup benchmarking, I first ran many tests with a variety of search densities namely, and 56 tests. The results of those tests were somewhat inconclusive, so I fixed the values of the seed and dungeon size to more accurately depict the effects of density on speedup. From the latter graphs, it seemed that there is a common local maximum around a density of **0.25,** despite my previous tests being somewhat misleading.

Because of this I used a fixed density of **0.25** within my benchmarking of my parallel program against the serial version.

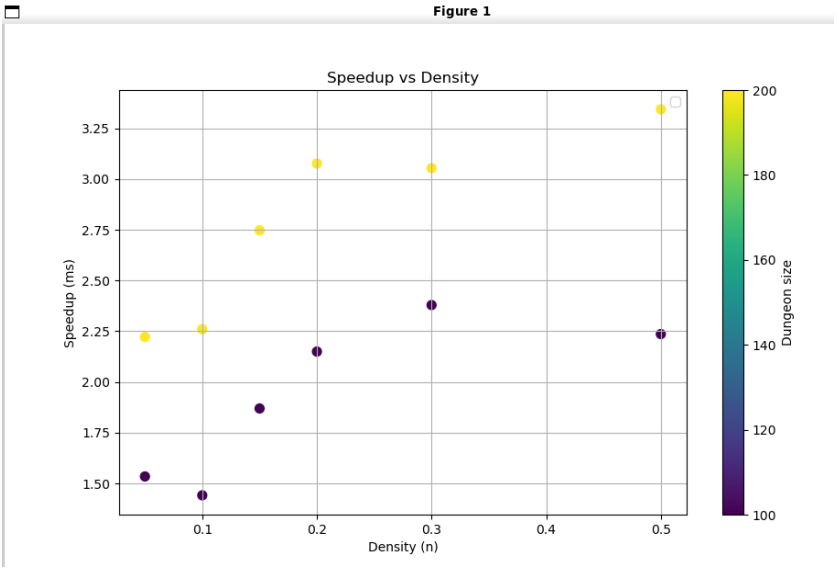
Previous results:



Second Testing Results:

Test values:

size	density	seed
100	0.05	42
100	0.1	42
100	0.15	42
100	0.2	42
100	0.3	42
100	0.5	42
200	0.05	42
200	0.1	42
200	0.15	42
200	0.2	42
200	0.3	42
200	0.5	42



Validation:

For the validation of my parallel solution, I created a python program, ***CompareImages.py***, which can compile, run and clean both my parallel solution and the serial solution files within its directory (my Validation and efficiency folder). ***CompareImages.py*** works by checking that the

number of searches as well as the x and y boss location of both the serial and parallel programs are identical, either Passing or Failing in the case of the coordinates or returning a zero number of searches if the search number is not the same. Critically it checks that the path pictures, ***visualiseSearchPath.png***, are identical pixel-wise by making use of pythons ***ImageChops*** module which allows for performing arithmetical operations on images. By taking the difference of the two images for each test within my ***testValues.csv*** file, if the resulting image after the subtraction is just black that implies that the images are identical and the test is passed, and if not, the test is failed.

Code snippet from CompareImages.py:

```python
def same_image(pic1_path, pic2_path):
    """
    Check if the serial and parallel images are the same
    """
    pic1 = Image.open(pic1_path)
    pic2 = Image.open(pic2_path)

    # Subtracting 1 image from the other image: if the result is completely black -> identical
    diff = ImageChops.difference(pic1, pic2)
    return not diff.getbbox()
#_____________________________________________________________________

def format_output(output_text):
    """
    Formatting a text file so its easier to plot my speed up graphs:
    format: Test number, DungeonSize, Density, Seed, Serial time (ms), SerialGridPoints, Parallel time (ms) ParallelGridPoints, NumSearches, Image comparison, Coords Match
    """
    time_match = re.search(r'time:\s*(\d+)\s*ms', output_text)
    points_match = re.search(r'number dungeon grid points evaluated:\s*(\d+)', output_text)
    searches_match = re.search(r'Number searches:\s*(\d+)', output_text)
    coord_match = re.search(r'Dungeon Master.*x=([\d\.\-]+)\s*y=([\d\.\-]+)', output_text)

    time_ms = int(time_match.group(1)) if time_match else 0
    points = int(points_match.group(1)) if points_match else 0
    num_searches = int(searches_match.group(1)) if searches_match else 0
    x = float(coord_match.group(1)) if coord_match else None
    y = float(coord_match.group(2)) if coord_match else None

    return time_ms, points, num_searches, (x, y)
#
```

For both the validation and benchmarking of my parallel solution I used the same test values from a csv file, **testValues.csv**. which consists of **76** test cases with dungeon sizes that range from **50-640** for two different seed **24** and **121**. For the interest of saving space in this report however I'll only display the top 16 results:

dungeon_size	num_searches	seed	description
50	0.25	24	Very small grid seed 24
60	0.25	24	Small grid seed 24
70	0.25	24	Small-medium grid seed 24
80	0.25	24	Medium grid seed 24
90	0.25	24	Medium grid variant seed 24
100	0.25	24	Standard grid seed 24
110	0.25	24	Standard grid variant seed 24
120	0.25	24	Medium-large grid seed 24
130	0.25	24	Medium-large variant seed 24
140	0.25	24	Grid size 140 seed 24
150	0.25	24	Grid size 150 seed 24
160	0.25	24	Grid size 160 seed 24
170	0.25	24	Grid size 170 seed 24
180	0.25	24	Grid size 180 seed 24
190	0.25	24	Grid size 190 seed 24
200	0.25	24	Grid size 200 seed 24

Etc.

First 16 (out of 76) test results (ComparisonResults.csv):

Test_number	DungeonSize	Density	Seed	SerialTime_(ms)	SerialGridPoints	ParallelTime_(ms)	ParallelGridPoints	NumSearches	ImageComparison	CoordComparison
1	50	0.25	24	68	172473	47	151444	12500	PASS	PASS
2	60	0.25	24	82	242601	60	205177	18000	PASS	PASS
3	70	0.25	24	117	326473	61	220897	24500	PASS	PASS
4	80	0.25	24	126	424820	60	242666	32000	PASS	PASS
5	90	0.25	24	205	542386	61	243879	40500	PASS	PASS
6	100	0.25	24	209	672743	81	297615	50000	PASS	PASS
7	110	0.25	24	222	804084	100	383952	60500	PASS	PASS
8	120	0.25	24	301	954166	123	444530	72000	PASS	PASS
9	130	0.25	24	417	1138363	138	529159	84500	PASS	PASS
10	140	0.25	24	379	1308619	163	625273	98000	PASS	PASS
11	150	0.25	24	465	1501931	168	693861	112500	PASS	PASS
12	160	0.25	24	493	1712361	170	649704	128000	PASS	PASS
13	170	0.25	24	583	1944754	200	724477	144500	PASS	PASS
14	180	0.25	24	620	2176226	233	945598	162000	PASS	PASS
15	190	0.25	24	727	2408336	263	1020363	180500	PASS	PASS
16	200	0.25	24	822	2664268	271	1058951	200000	PASS	PASS

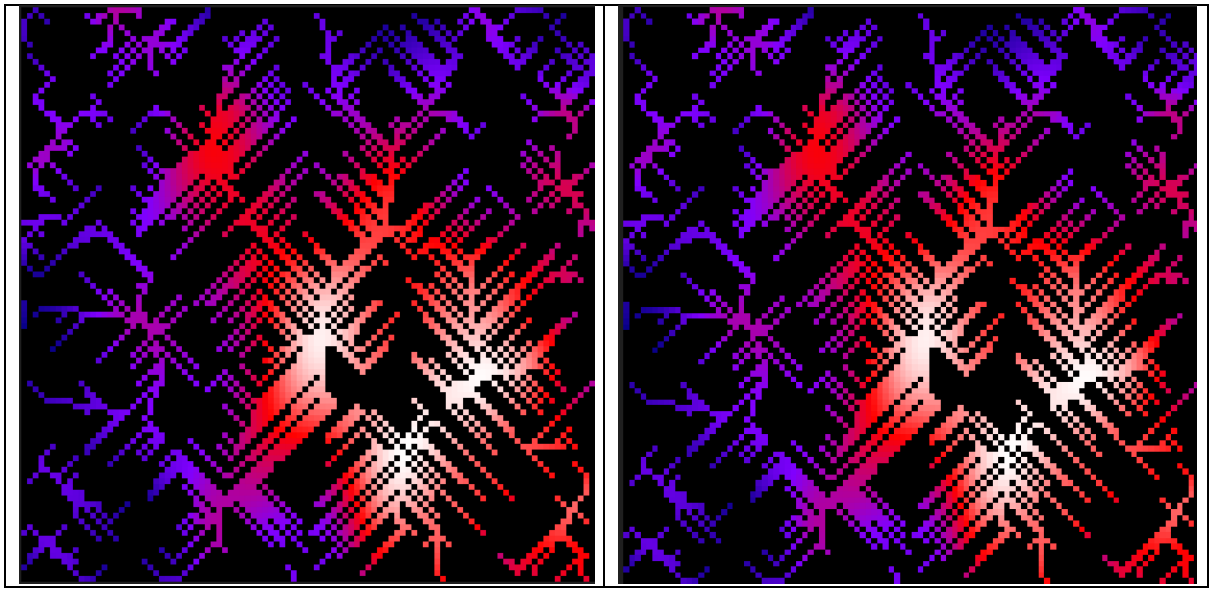
Etc.

It is clear from the PASS test results that my program was correctly implemented however I wanted to double check because before the bug fixes were released and before I created my validation programs, the images my program created were very similar but not identical, sometimes only 1 pixel apart. I didn't want to have the same issue again, so I checked a few of the tests manually to make sure that the images produced were in fact the same.

Eg:

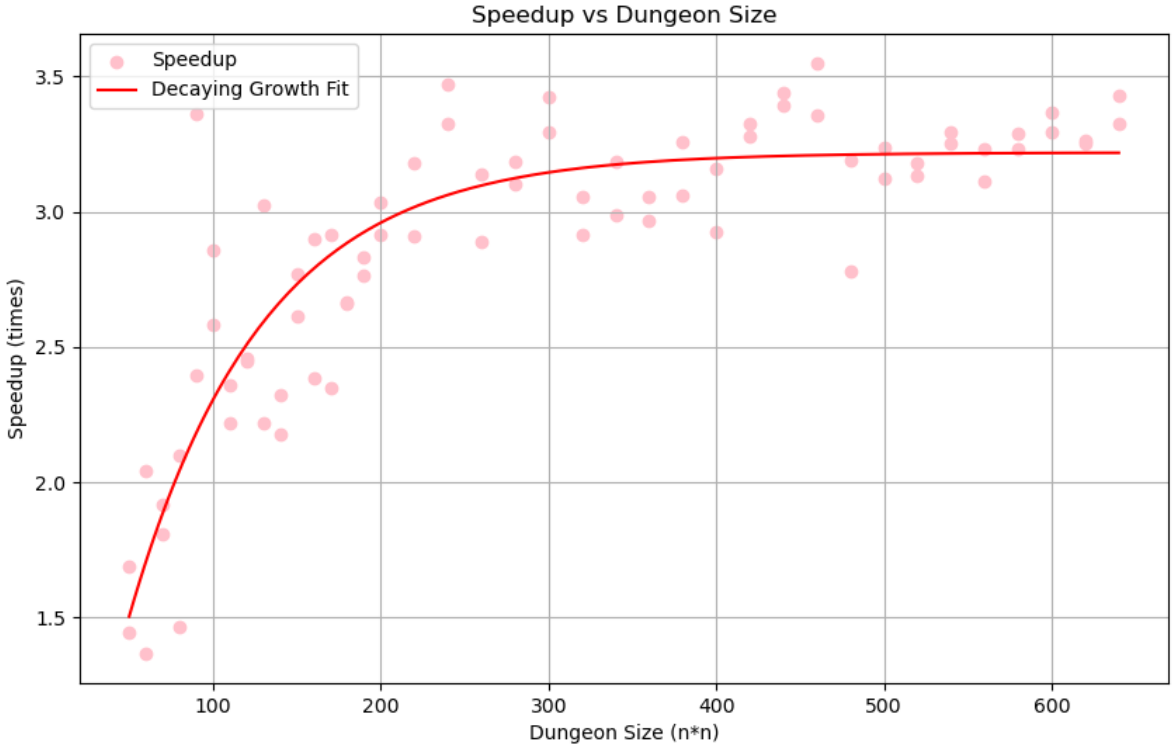
Serial Solution	**Parallel Solution**
ARGS ?= 10 0.25 99	ARGS ?= 10 0.25 99

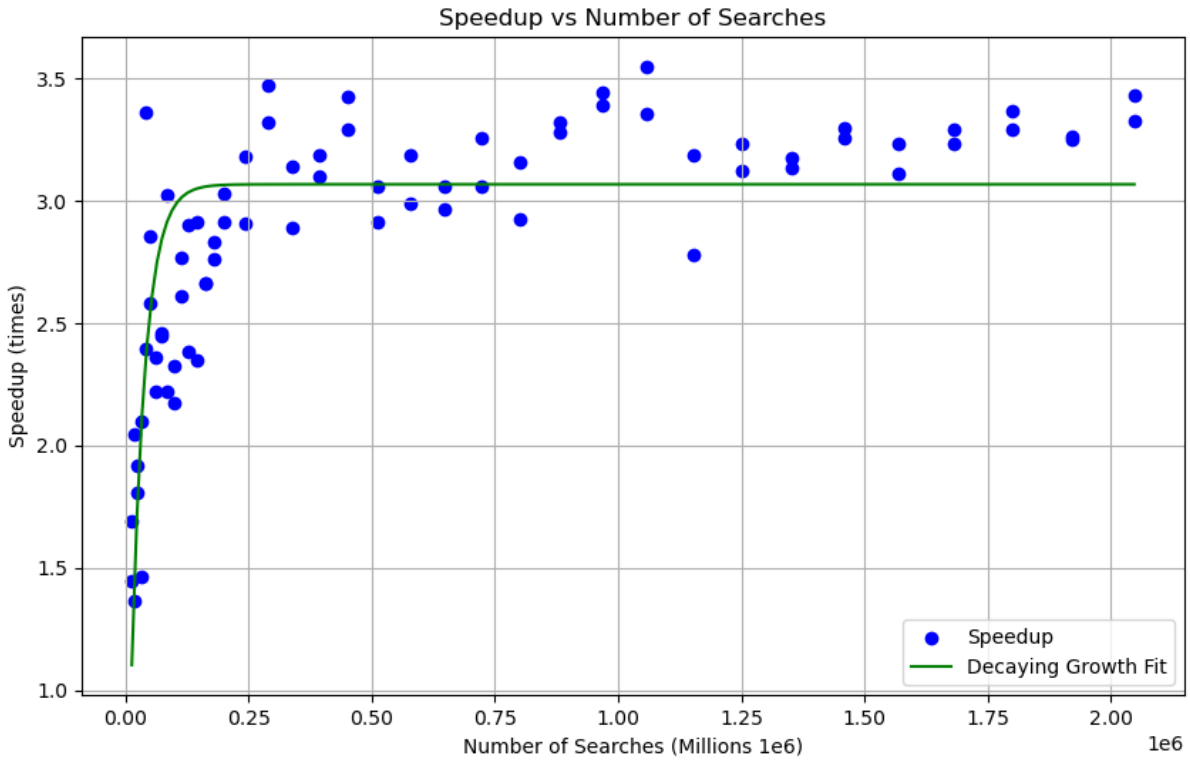
`java -cp SoloLevelling DungeonHunter 10 0.25 99`
`dungeon size: 10,`
`rows: 100, columns: 100`
`x: [-10.000000, 10.000000], y: [-10.000000, 10.000000]`
`Number searches: 500`
 	
`time: 5 ms`
`number dungeon grid points evaluated: 7301 (73%)`
`Dungeon Master (mana 116915) found at: x=3.2 y=-5.6`
	`java DungeonHunterParallel 10 0.25 99`
	`dungeon size: 10,`
	`rows: 100, columns: 100`
	`x: [-10.000000, 10.000000], y: [-10.000000, 10.000000]`
	`Number searches: 500`
	
	`time: 5 ms`
	`number dungeon grid points evaluated: 7301 (73%)`
	`Dungeon Master (mana 116915) found at: x=3.2 y=-5.6`



Benchmarking:

Using the 76 test results from my ***ComparisonResults.csv*** file where the density is fixed at **0.25**, since the number of searches depends on density and grid size, fixing density makes dungeon size the sole factor (**size: 50-640**) and the results are adequate for generating both the Size vs. Speedup and Num_Searches vs. Speedup graphs. I graphed my results using a python program I created called ***Graphing.py*** which generates scatterplots for both and, with the help of ChatGPT, I added a regression line on top which follows the exponential decay function. The threshold value within my program was **5000**.





Discussion:

My Parallel Program Performs best for large inputs since the cost of evaluating each grid point is high and there is a large workload which benefits from being divided, however, returns start capping off for grids larger than **300*300** pixels and more than **~125 million** searches and the speed-up limits at around **3.5x**. The maximum speedup achieved was **3.54x** for a dungeon size of **460** and **1058000** searches. Since the departmental server runs on **4** cores (so I've been told at least) to achieve ideal speedup, linear speedup, my maximum speedup would've had to have been **4x**. Although I did not reach this linear ideal, as expected by Amdahl's law, I did get very close as it is **88.5%** ideal:

$$\left(\frac{3.54}{4}\right) * 100\% = 88.5\%$$

While running my tests however I was streaming shows and completing other work since I had so many test cases. This concurrent use of my laptops resources may have affected the speedup results as resources got split and potentially skewed the values. However, I tried to combat this by having such a large sample size in the first place. From my sample data, there appears to be very few outlying cases, and most data points follow the fitted curve with slight variance. The consistent scaling suggests that the workload was quite evenly balanced between threads and the system environment during testing.

Conclusion:

My average speed up across tests was **2.87x,** this speedup is **71.75%** ideal. This is a very significant performance gain and coupled with the fact that the inherent nature of performing thousands of independent searches on a grid is embarrassingly parallel, this serial code of the assignment was a prime candidate for parallelization. From the benchmarking results it is worth

using parallelization particularly for large problem/grid sizes upwards of **100*100** grids or **10000** searches especially since the implementation of he fork/join framework offered a very clean solution.
