Methods:

Parallelization approach:

The main idea behind parallelizing this scenario is dividing the search operations evenly among available threads using the Fork/Join framework. Each thread was responsible for performing searches on a subset of the grid points. Parallelization was used in this scenario by using a fork join pool to recursively split up the array of search objects into smaller arrays of search objects until a sequential cutoff was reached. A new class called “ResultOfSearch” was also created, which was used in conjunction with Recursive Task. The “ResultOfSearch” class keeps track of both the minimum height and the index of the search object that found it.

Issues and Considerations:

The main “issue” encountered was that the parallel solution would sometimes return different x,y coordinates for the global minimum height compared to the serial solution, but always the same minimum height. This could be due to there being a possibility of the same minimum height at different points due to the size of the terrain grid, and the threads ended on and returned that different point. For considerations, an adequate sequential cutoff was decided on and used to ensure each thread performed a reasonably balanced amount of work. This cutoff ensures that if a search task is too small, it's executed sequentially instead of creating unnecessary overhead.

Optimizations:

To optimize the parallel solution, there was testing of different sequential cutoff values ranging from 500-5000 which ultimately was settled on 1000, which provided the all around fastest times. Also when recording times and values, the program was run 5 times, the 2 highest times were discarded and then the median time of the remaining 3 was selected. Also, when using the Rosenbrock function to check for validation, there were no negative numbers included for the range of the x,y values. It seems that when including negative numbers it doesnt give the intended results of having a global minimum of 0 at x = 1, y = 1, which is used for testing the “correctness”. However, the Rosenbrock is subject to randomness and that could be what was causing the different results.

Validation:

Serial Solution:

Rows	Columns	X Range	Y Range	Search Density	Global Minimum
100	100	-100;100	-100;100	1	-32295 at x=80,0 y=6,0
500	500	-500;500	-500;500	1	-32416 at x=344,0 y=6,0
1000	1000	-1000;1000	-1000;1000	1	-32416 at x=-366,0 y=6,0
3000	3000	-3000;3000	-3000;3000	1	-32416 at x=-2496,0 y=6,0
5000	5000	-5000;5000	-5000;5000	0.5	-32416 at x=1764,0 y=6,0
8000	8000	-8000;8000	-8000;8000	1	-32416 at x=-366,0 y=6,0

Parallel Solution:

Rows	Columns	X Range	Y Range	Search Density	Global Minimum
100	100	-100;100	-100;100	1	-32295 at x=80,0 y=6,0
500	500	-500;500	-500;500	1	-32416 at x=344,0 y=6,0
1000	1000	-1000;1000	-1000;1000	1	-32416 at x=344,0 y=6,0
3000	3000	-3000;3000	-3000;3000	1	-32416 at x=2474,0 y=6,0
5000	5000	-5000;5000	-5000;5000	0.5	-32416 at x=1764,0 y=6,0
8000	8000	-8000;8000	-8000;8000	1	-32416 at x=-366,0 y=6,0

Validation using The Rosenbrock function:

Serial Solution (Using no negative x,y range values):

Rows	Columns	X Range	Y Range	Search Density	Global Minimum
100	100	0;100	0;100	1	0 at x=1,0 y=1,0
500	500	0;500	0;500	1	0 at x=1,0 y=1,0
1000	1000	0;1000	0;1000	1	0 at x=1,0 y=1,0
3000	3000	0;3000	0;3000	1	0 at x=1,0 y=1,0
5000	5000	0;5000	0;5000	0.5	0 at x=1,0 y=1,0
8000	8000	0;8000	0;8000	1	0 at x=1,0 y=1,0

Parallel Solution (Using no negative x,y range values):

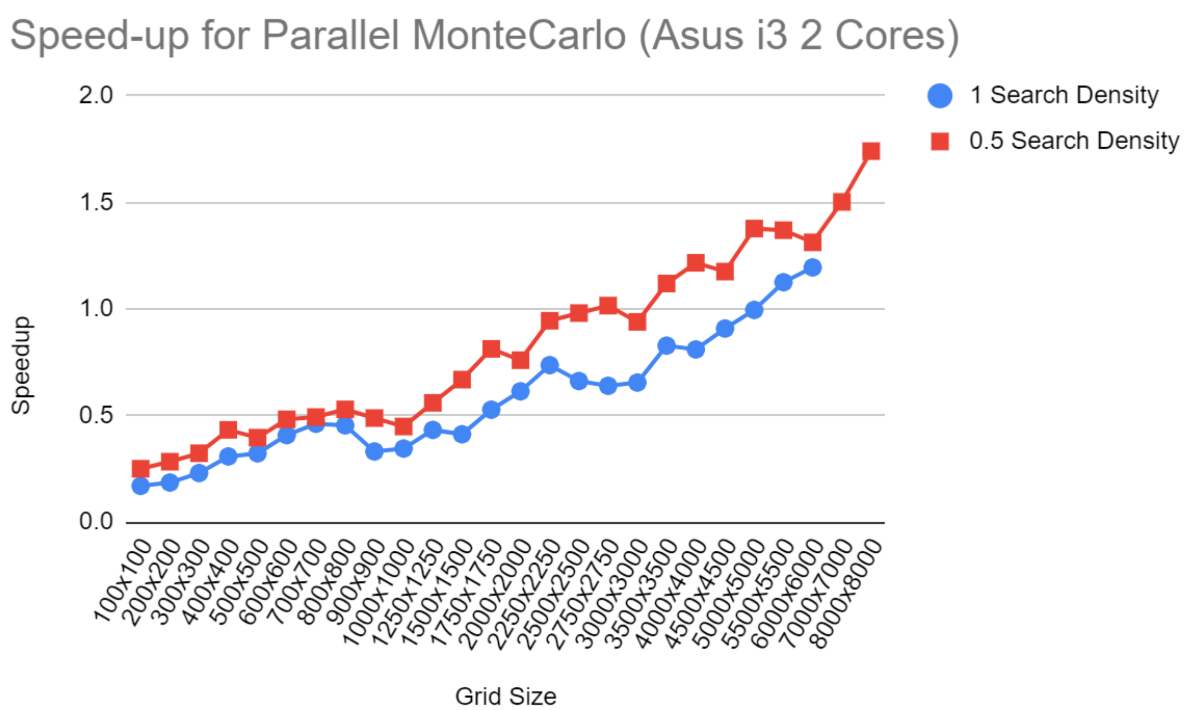
Rows	Columns	X Range	Y Range	Search Density	Global Minimum
100	100	0;100	0;100	1	0 at x=1,0 y=1,0
500	500	0;500	0;500	1	0 at x=1,0 y=1,0
1000	1000	0;1000	0;1000	1	0 at x=1,0 y=1,0
3000	3000	0;3000	0;3000	1	0 at x=1,0 y=1,0
5000	5000	0;5000	0;5000	0.5	0 at x=1,0 y=1,0

8000	8000	0;8000	0;8000	1	0 at x=1,0 y=1,0

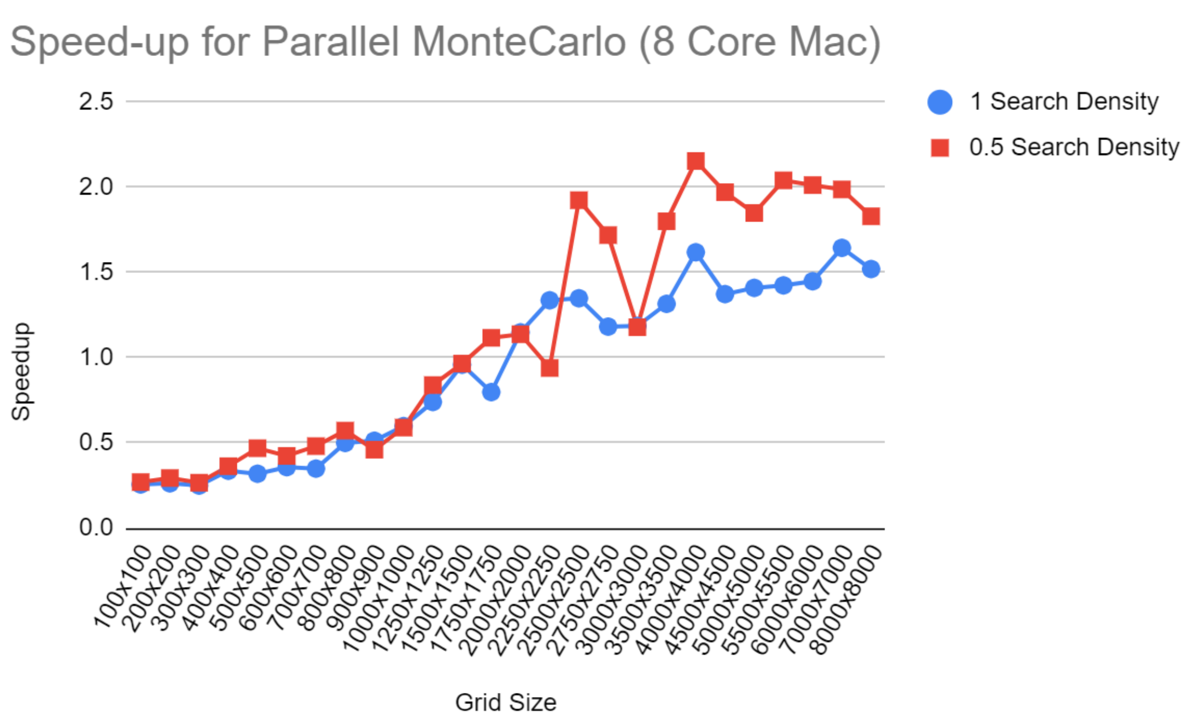
Benchmarking:

Grid Sizes	Median Parallel Time(8 Cores,1 Search Density)	Median Parallel Time (8 Cores,0.5 Search Density)	Median Parallel Time (2 Cores, 0.5 Search Density)	Median Parallel Time (2 Cores, 1 Search Density)
100x100	39	37	40	59
200x200	57	51	53	81
300x300	80	75	62	87
400x400	95	88	74	104
500x500	119	81	96	118
600x600	148	125	110	130
700x700	186	135	132	141
800x800	166	145	157	183
900x900	193	215	203	299
1000x1000	202	205	270	351
1250x1250	262	231	347	449
1500x1500	294	292	422	685
1750x1750	526	376	517	797
2000x2000	516	521	781	967
2250x2250	584	831	827	1061
2500x2500	779	546	1072	1589
2750x2750	1111	764	1294	2056
3000x3000	1302	1310	1645	2361
3500x3500	1706	1247	2007	2714
4000x4000	1907	1433	2539	3814
4500x4500	2848	1984	3329	4309
5000x5000	3518	2682	3603	4985
5500x5500	4206	2936	4377	5327
6000x6000	4956	3567	5473	6012
7000x7000	5949	4923	6516	crashed
8000x8000	8407	6985	7347	crashed

Results:



Speed-up for Parallel MonteCarlo (Asus i3 2 Cores)
1 Search Density
0.5 Search Density
Speedup
2.0
1.5
1.0
0.5
0.0
100x100
200x200
300x300
400x400
500x500
600x600
700x700
800x800
900x900
1000x1000
1250x1250
1500x1500
1750x1750
2000x2000
2250x2250
2500x2500
2750x2750
3000x3000
3500x3500
4000x4000
4500x4500
5000x5000
5500x5500
6000x6000
7000x7000
8000x8000
Grid Size



Speed-up for Parallel MonteCarlo (8 Core Mac)
1 Search Density
0.5 Search Density
Speedup
2.5
2.0
1.5
1.0
0.5
0.0
100x100
200x200
300x300
400x400
500x500
600x600
700x700
800x800
900x900
1000x1000
1250x1250
1500x1500
1750x1750
2000x2000
2250x2250
2500x2500
2750x2750
3000x3000
3500x3500
4000x4000
4500x4500
5000x5000
5500x5500
6000x6000
7000x7000
8000x8000
Grid Size

For what range of grid sizes does your parallel program perform well:

It is quite clear that the parallel program starts performing better with increasing grid sizes and for grids with small ranges that do not require much computing; parallelizing the program has the inverse effect intended and increases the computation time. Although the parallel program does not achieve a great speedup at lower grid sizes, this makes sense as the action of recursively splitting up the search object array and handing the process to different processors would ultimately be slower than just simply doing it sequentially. Around the grid size of 2000 x 2000 the 8 core machine architectures start achieving more than 1 times speedup for both search densities and at around 3000 x 3000 the 2 core machine starts achieving speedup for the 0.5 search density. For the 2 core machine architecture at 1 search density there is only a slight speedup achieved from 5000 x 5000 onwards. If the grid sizes were to keep increasing along with the number of cores, the parallel program would keep getting better speedup times compared to the serial program. Unfortunately the 2 core machine architecture could not run any grid size greater than 6000 x 6000 hence the missing 2 data points. Overall, the fastest speed up was achieved on the machine architecture with more cores and with the search density being the lowest. This makes sense as having more processors allows the program to be sped up more and the lower amount of searches performed will result in a quicker runtime. The worst speedup achieved was on the machine architecture with 2 cores and the higher search density. Once again this makes sense as having less cores will only slow down the program and having the higher search density will increase the amount of searches performed and increase the runtime.

What is the maximum speedup you obtained and how close is this speedup to the ideal expected:

There was a maximum speedup of 2.1 for 0.5 search density and 1.6 for 1 search density achieved on the 8 core machine architecture. For the 2 core machine architecture there was a maximum speed up of 1.7 for 0.5 search density and 1.2 for 1 search density achieved. Having a lower speedup for a higher search density makes sense because you are increasing the number of searches you are doing on the grid which will increase the time of the program. According to Amdahl's Law, ideal speed-up = 1/((1-P) + [P-N]) where P is the proportion of the program that can be parallelized and N is the number of processes. Due to the entire program being able to be parallelized, the ideal speed-up on the 8-core machine is 8 and the ideal speed-up on the 2-core machine is 2.

How reliable are your measurements:

For each grid size tested for times, the program was run 5 times, the 2 highest times were discarded and the median time of the remaining 3 times was chosen. There was an announcement that said that this method of recording times was sufficient for this assignment and that it was better than just merely taking averages, which are heavily influenced by outliers and wouldn't be a sufficient representation of the true time.

Are there any anomalies and can you explain why they occur:

There aren't really any big anomalies or outliers in the data collected. There are 2 missing data points but that is simply due to the 2 core machine not being able to run a test with grid size parameters greater than 6000 x 6000. The data follows a general positive upward trend and some spikes are to be expected due the randomness of the program. Some anomalies could have occurred if it was decided to rather take the average of all the times which is vulnerable to outliers and could result in some larger or smaller times.

Conclusions:

In reference to this MonteCarlo minimization program, parallelizing is a worthwhile process as the input parameters fed to the program can be large and lead to extensive computational needs. Obviously for smaller grid sizes the serial solution is the better option because simply doing it sequentially for those smaller grid sizes is better than doing the whole recursive process of splitting of the search object array and handing it to different processors. This program and test parameters were used as a basis for an assignment for University and scaled down, but if this was done in the real world, running this simulation for MUCH greater parameters then parallelization would be very useful and would definitely be chosen over a simple sequential way of programming.