ECHKEN001:

Kent Echardt

CSC2002S Assignment PCP1 2023

Parallel Programming with Java:

Parallelizing Monte Carlo Function Optimisation:

1. Parallelization approach:

When trying to parallelize the serial algorithm of the Monte Carlo method, I essentially used the process of finding the global minimum across all the grids by looping through all the number of searches (Area of rows * columns) and finding the minimum function and put that whole process into a separate method called compute() for my ForkJoinPool I can create and invoke. Before I invoke the ForkJoinPool method, I created an constructor called the name of the class "MonteCarloMinimizationParallel" with the parameters being the start and end index and the array of searches, then in the main method, I created the object called MonteCarloMinimizationParallel with the parameters of the search array, an start and end index integer with the start being 0 and the end index being the actual number of searches.

Then I created the ForkJoinPool object with the parameters being the object of MonteCarloMinimizationParallel and I invoked the method which led to computing the process in compute(). The idea is to split the search workload into smaller subtasks that can be executed concurrently, and then combine the results to find the overall global minimum.

In the compute method, you start by checking if the range of searches (end - start) is smaller than the threshold value. If the range is smaller than the threshold, the number of searches which is the task is considered small enough to be processed in a sequential manner. Otherwise the tasks is split into two subtasks, "left" and "right" halves. Each subtask is created with a narrower range of searches (start to split and then split to end in order) since it recursively divides by 2 and split in half. These subtasks are then forked and executed concurrently.

The left and right halves of the range, are executed in parallel by different threads. Each subtask follows a similar logic to the sequential approach: it iterates through its assigned range of searches and finds the local minimum in each grid. The left subtask is forked using the fork method, and then the right subtask is computed sequentially using the current thread, after the right subtask completes, the main thread waits for the left subtask to complete using the join method so that the results from both subtasks are combined correctly.

Particular Issues/Considerations:

It would be a problem if I implemented the methods from the inherited java.lang.Threads class as the methods there would not suit the complex process of dividing the search workload into smaller subtasks, then forking one half of the tasks, then computing the other half, then the main thread to finally wait until the left subtasks is completed so that it ensures that both subtasks finish before proceeding.

So when dividing the work among threads, it's important to maintain load balance. Uneven distribution of work can lead to some threads finishing early while others are still working, which will cause subpar performance. In the compute() method, we split the work into two subtasks of equal size, helping to ensure a more balanced distribution of computation.

The parallelization introduces the need for sharing data, such as the searches array and the min and finder variables, among threads. Proper synchronization mechanisms, like locks or synchronization blocks, are necessary to prevent race conditions and ensure the accuracy of the results.

Optimization:

The compute() method first checks if the total number of searches is below the CUTOFF. If this condition is met, the method performs the searches sequentially. This optimization prevents unnecessary thread creation for small tasks, which could lead to increased overhead.

The splitting of work into subtasks is performed by dividing the search space exactly in half. This balanced splitting ensures that both sides of the search space have the same workloads, promoting equal load balance among threads.

I’ve used the ForkJoinPool method to manage the execution of the parallel tasks instead of the Thread class methods. The pool automatically manages the thread pool and the allocation of tasks to threads.

The decision to process a task in parallel or sequentially depends on the range of searches (end - start). If the range is small enough, it's processed sequentially; otherwise, it's split into subtasks for parallel processing.

Validation and How I benchmarked my algorithm:

When I tested out the tested out my algorithm, I both compiled and ran my serial and parallel programs and saw if it had the same global minimum, with the same arguments for the parameters on both of those, and they were always the same, and I had to make sure they stayed the same.

I tested the algorithms on values of the parameters which the grid sizes (rows * columns) increase by 1000 for every test keeping every other values constant, and another series of tests which increases the search density by 0.1 for every test, seeing as those two variables facilitate the most change in the time in ms.

I benchmarked both results from nightmare and the personal computer with the same exact arguments to see if there was any difference in the speedup.

Problems I’ve encountered: One of the biggest difficulties was seeing that the x-coordinates and the y-coordinates of the terrain object changed both in the serial and parallel version of the program, but it had the same global minimum, so it was difficult trying to make out if the parallelisation was actually working.

Benchmark Results:

Running the program on 2 architectures:

1. Nightmare.cs.uct.za: 8 cores

2. Personal Computer: Laptop: 4 cores
   Processor: AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx, 2100 Mhz, 4 Core(s), 8 Logical Processor(s)
   OS Name: Microsoft Windows 11 Home Single Language; Version: 10.0.22621 Build 22621
   System Type:x64-based PC

Nightmare: Parallel

Arguments: Starting at Rows: 1000

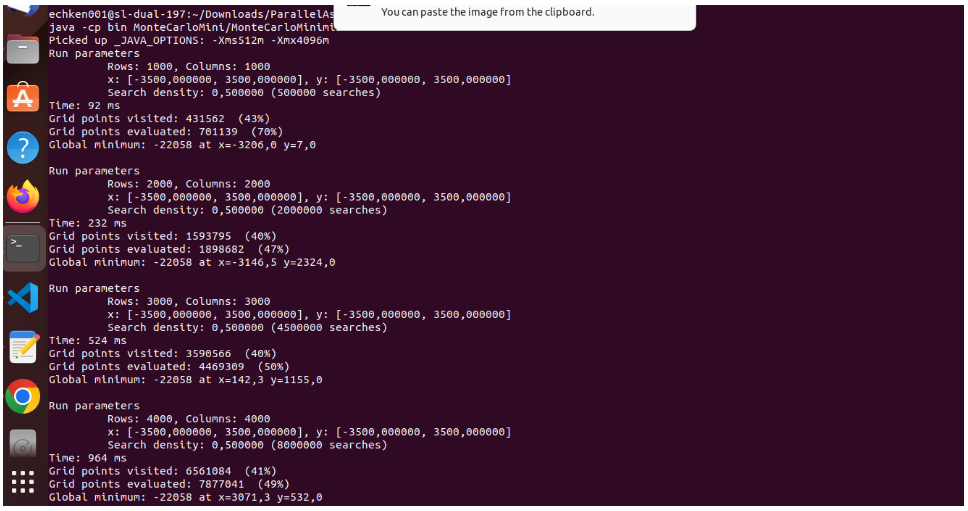
Starting at Columns: 1000

Increasing by 1000 on both rows and columns for every result

Xmin: -3500; Xmax: 3500

Ymin: -3500; Ymax: 3500

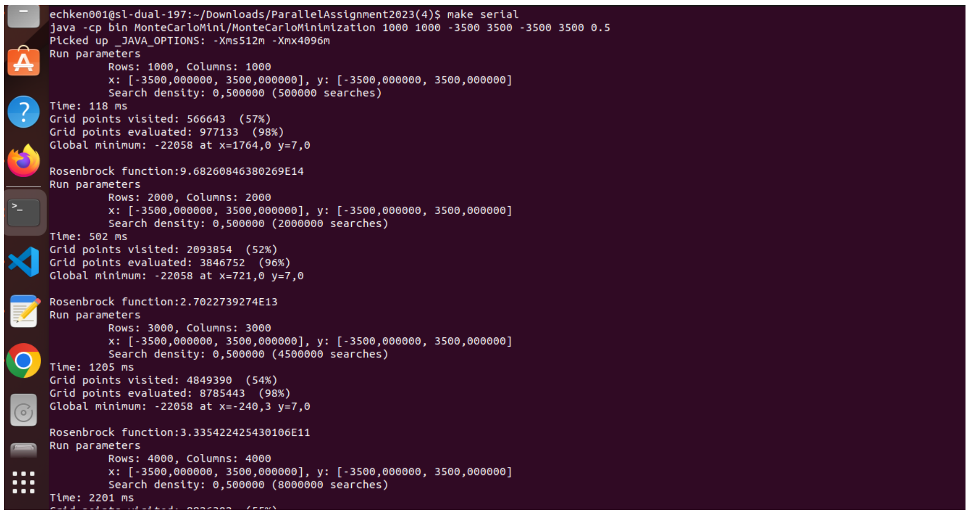
Search-Density: 0.5



```
echken001@sl-dual-197:~/Downloads/ParallelAs
java -cp bin MonteCarloMini/MonteCarloMinimi
Picked up _JAVA_OPTIONS: -Xms512m -Xmx4096m
Run parameters
         Rows: 1000, Columns: 1000
         x: [-3500,000000, 3500,000000], y: [-3500,000000, 3500,000000]
         Search density: 0,500000 (500000 searches)
Time: 92 ms
Grid points visited: 431562  (43%)
Grid points evaluated: 701139  (70%)
Global minimum: -22058 at x=-3206,0 y=7,0

Run parameters
         Rows: 2000, Columns: 2000
         x: [-3500,000000, 3500,000000], y: [-3500,000000, 3500,000000]
         Search density: 0,500000 (2000000 searches)
Time: 232 ms
Grid points visited: 1593795  (40%)
Grid points evaluated: 1898682  (47%)
Global minimum: -22058 at x=-3146,5 y=2324,0

Run parameters
         Rows: 3000, Columns: 3000
         x: [-3500,000000, 3500,000000], y: [-3500,000000, 3500,000000]
         Search density: 0,500000 (4500000 searches)
Time: 524 ms
Grid points visited: 3590566  (40%)
Grid points evaluated: 4469309  (50%)
Global minimum: -22058 at x=142,3 y=1155,0

Run parameters
         Rows: 4000, Columns: 4000
         x: [-3500,000000, 3500,000000], y: [-3500,000000, 3500,000000]
         Search density: 0,500000 (8000000 searches)
Time: 964 ms
Grid points visited: 6561084  (41%)
Grid points evaluated: 7877041  (49%)
Global minimum: -22058 at x=3071,3 y=532,0
```

Nightmare Serial:

Same Input



```
echken001@sl-dual-197:~/Downloads/ParallelAssignment2023(4)$ make serial
java -cp bin MonteCarloMini/MonteCarloMinimization 1000 1000 -3500 3500 -3500 3500 0.5
Picked up _JAVA_OPTIONS: -Xms512m -Xmx4096m
Run parameters
         Rows: 1000, Columns: 1000
         x: [-3500,000000, 3500,000000], y: [-3500,000000, 3500,000000]
         Search density: 0,500000 (500000 searches)
Time: 118 ms
Grid points visited: 566643  (57%)
Grid points evaluated: 977133  (98%)
Global minimum: -22058 at x=1764,0 y=7,0

Rosenbrock function:9.68260846380269E14
Run parameters
         Rows: 2000, Columns: 2000
         x: [-3500,000000, 3500,000000], y: [-3500,000000, 3500,000000]
         Search density: 0,500000 (2000000 searches)
Time: 502 ms
Grid points visited: 2093854  (52%)
Grid points evaluated: 3846752  (96%)
Global minimum: -22058 at x=721,0 y=7,0

Rosenbrock function:2.7022739274E13
Run parameters
         Rows: 3000, Columns: 3000
         x: [-3500,000000, 3500,000000], y: [-3500,000000, 3500,000000]
         Search density: 0,500000 (4500000 searches)
Time: 1205 ms
Grid points visited: 4849390  (54%)
Grid points evaluated: 8785443  (98%)
Global minimum: -22058 at x=-240,3 y=7,0

Rosenbrock function:3.335422425430106E11
Run parameters
         Rows: 4000, Columns: 4000
         x: [-3500,000000, 3500,000000], y: [-3500,000000, 3500,000000]
         Search density: 0,500000 (8000000 searches)
Time: 2201 ms
```

Personal Computer architecture: Serial

Arguments: Starting at Rows: 1000

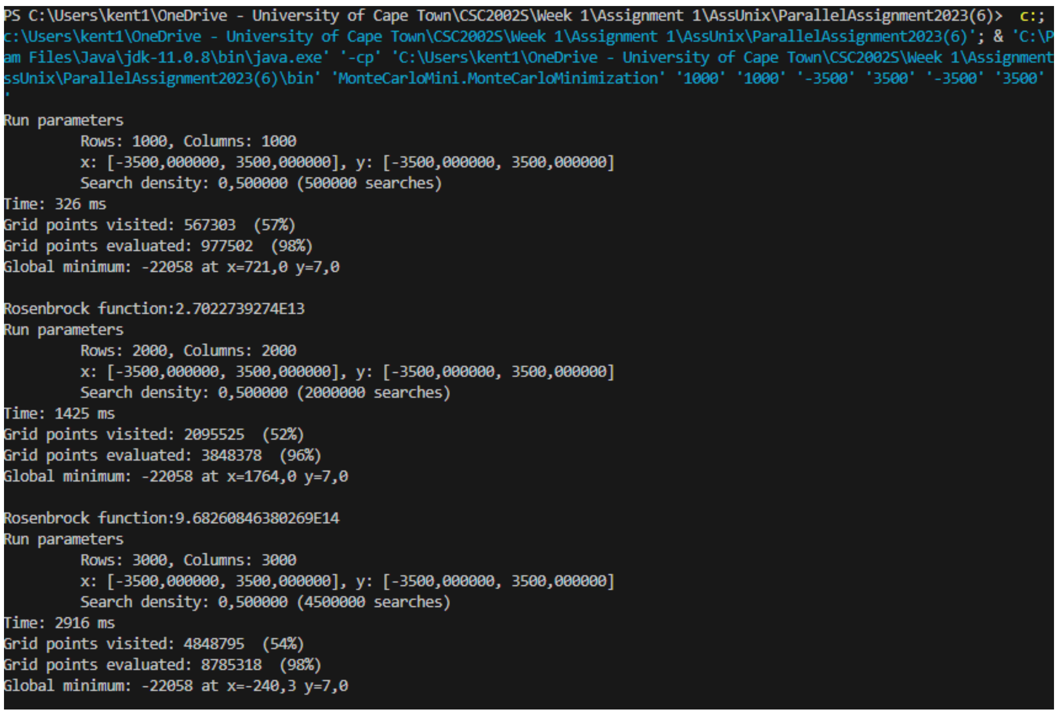
Starting at Columns: 1000

Increasing by 1000 on both rows and columns for every result

Xmin: -3500; Xmax: 3500

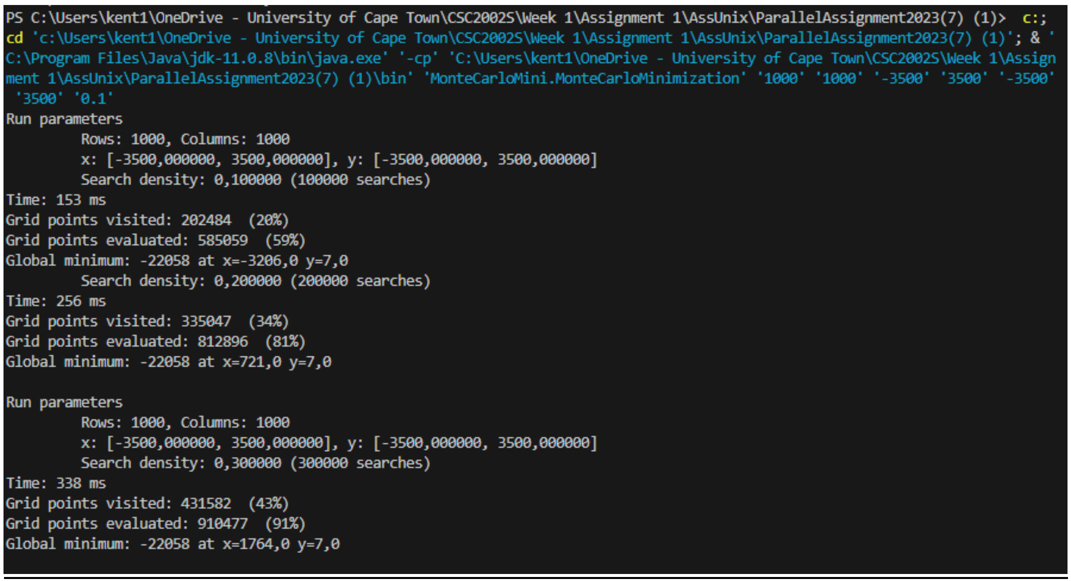
Ymin: -3500; Ymax: 3500

Search-Density: 0.5



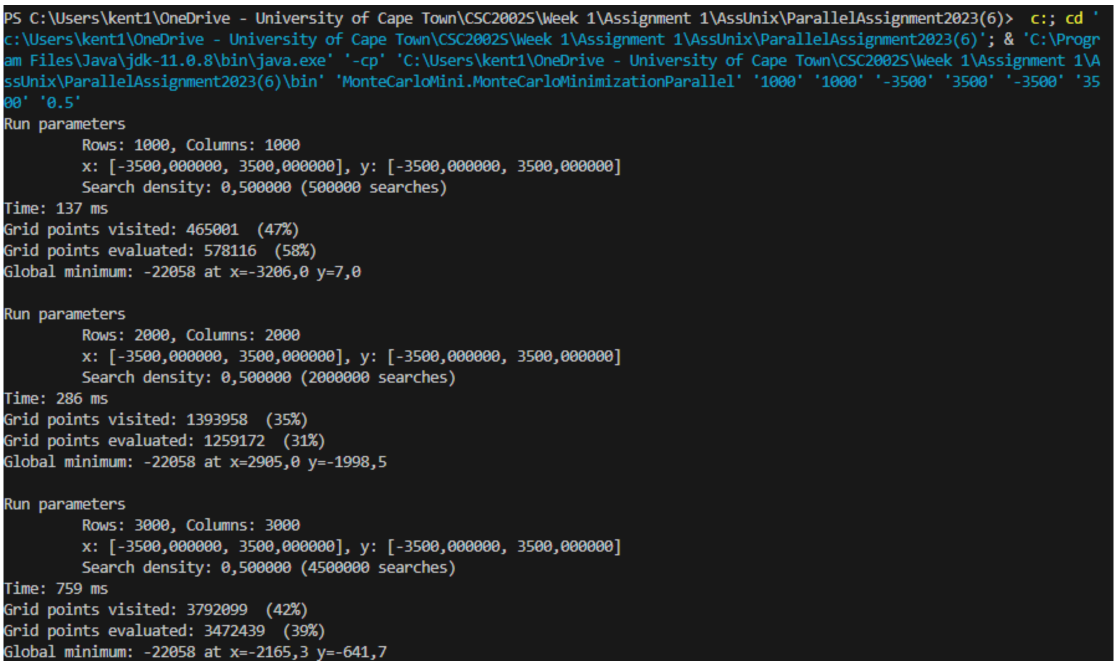
```
PS C:\Users\kent1\OneDrive - University of Cape Town\CSC2002S\Week 1\Assignment 1\AssUnix\ParallelAssignment2023(6)>  c:;
c:\Users\kent1\OneDrive - University of Cape Town\CSC2002S\Week 1\Assignment 1\AssUnix\ParallelAssignment2023(6)'; & 'C:\P
am Files\Java\jdk-11.0.8\bin\java.exe' '-cp' 'C:\Users\kent1\OneDrive - University of Cape Town\CSC2002S\Week 1\Assignment
ssUnix\ParallelAssignment2023(6)\bin' 'MonteCarloMini.MonteCarloMinimization' '1000' '1000' '-3500' '3500' '-3500' '3500'
'
Run parameters
        Rows: 1000, Columns: 1000
        x: [-3500,000000, 3500,000000], y: [-3500,000000, 3500,000000]
        Search density: 0,500000 (500000 searches)
Time: 326 ms
Grid points visited: 567303  (57%)
Grid points evaluated: 977502  (98%)
Global minimum: -22058 at x=721,0 y=7,0

Rosenbrock function:2.7022739274E13
Run parameters
        Rows: 2000, Columns: 2000
        x: [-3500,000000, 3500,000000], y: [-3500,000000, 3500,000000]
        Search density: 0,500000 (2000000 searches)
Time: 1425 ms
Grid points visited: 2095525  (52%)
Grid points evaluated: 3848378  (96%)
Global minimum: -22058 at x=1764,0 y=7,0

Rosenbrock function:9.68260846380269E14
Run parameters
        Rows: 3000, Columns: 3000
        x: [-3500,000000, 3500,000000], y: [-3500,000000, 3500,000000]
        Search density: 0,500000 (4500000 searches)
Time: 2916 ms
Grid points visited: 4848795  (54%)
Grid points evaluated: 8785318  (98%)
Global minimum: -22058 at x=-240,3 y=7,0
```

Increasing search density by 0.1 starting at 0.1 search density, keeping all other values constant



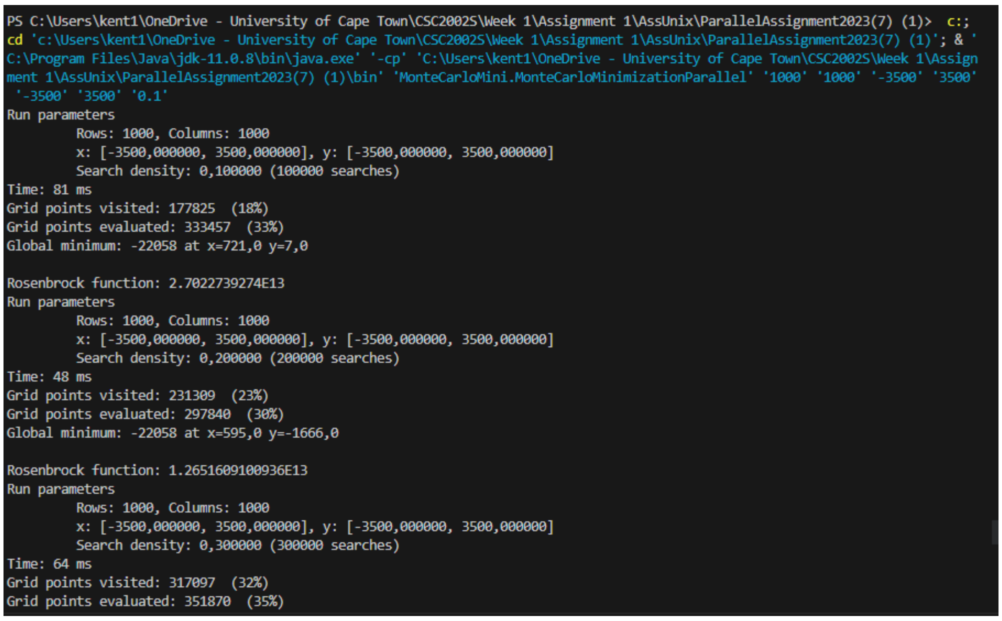
```
PS C:\Users\kent1\OneDrive - University of Cape Town\CSC2002S\Week 1\Assignment 1\AssUnix\ParallelAssignment2023(7) (1)>  c:;
cd 'c:\Users\kent1\OneDrive - University of Cape Town\CSC2002S\Week 1\Assignment 1\AssUnix\ParallelAssignment2023(7) (1)'; & '
C:\Program Files\Java\jdk-11.0.8\bin\java.exe' '-cp' 'C:\Users\kent1\OneDrive - University of Cape Town\CSC2002S\Week 1\Assign
ment 1\AssUnix\ParallelAssignment2023(7) (1)\bin' 'MonteCarloMini.MonteCarloMinimization' '1000' '1000' '-3500' '3500' '-3500'
 '3500' '0.1'
Run parameters
        Rows: 1000, Columns: 1000
        x: [-3500,000000, 3500,000000], y: [-3500,000000, 3500,000000]
        Search density: 0,100000 (100000 searches)
Time: 153 ms
Grid points visited: 202484  (20%)
Grid points evaluated: 585059  (59%)
Global minimum: -22058 at x=-3206,0 y=7,0
        Search density: 0,200000 (200000 searches)
Time: 256 ms
Grid points visited: 335047  (34%)
Grid points evaluated: 812896  (81%)
Global minimum: -22058 at x=721,0 y=7,0

Run parameters
        Rows: 1000, Columns: 1000
        x: [-3500,000000, 3500,000000], y: [-3500,000000, 3500,000000]
        Search density: 0,300000 (300000 searches)
Time: 338 ms
Grid points visited: 431582  (43%)
Grid points evaluated: 910477  (91%)
Global minimum: -22058 at x=1764,0 y=7,0
```

Personal computer architecture: Same Values: Parallel:



```
PS C:\Users\kent1\OneDrive - University of Cape Town\CSC2002S\Week 1\Assignment 1\AssUnix\ParallelAssignment2023(6)>  c:; cd '
c:\Users\kent1\OneDrive - University of Cape Town\CSC2002S\Week 1\Assignment 1\AssUnix\ParallelAssignment2023(6)'; & 'C:\Progr
am Files\Java\jdk-11.0.8\bin\java.exe' '-cp' 'C:\Users\kent1\OneDrive - University of Cape Town\CSC2002S\Week 1\Assignment 1\A
ssUnix\ParallelAssignment2023(6)\bin' 'MonteCarloMini.MonteCarloMinimizationParallel' '1000' '1000' '-3500' '3500' '-3500' '35
00' '0.5'
Run parameters
         Rows: 1000, Columns: 1000
         x: [-3500,000000, 3500,000000], y: [-3500,000000, 3500,000000]
         Search density: 0,500000 (500000 searches)
Time: 137 ms
Grid points visited: 465001  (47%)
Grid points evaluated: 578116  (58%)
Global minimum: -22058 at x=-3206,0 y=7,0

Run parameters
         Rows: 2000, Columns: 2000
         x: [-3500,000000, 3500,000000], y: [-3500,000000, 3500,000000]
         Search density: 0,500000 (2000000 searches)
Time: 286 ms
Grid points visited: 1393958  (35%)
Grid points evaluated: 1259172  (31%)
Global minimum: -22058 at x=2905,0 y=-1998,5

Run parameters
         Rows: 3000, Columns: 3000
         x: [-3500,000000, 3500,000000], y: [-3500,000000, 3500,000000]
         Search density: 0,500000 (4500000 searches)
Time: 759 ms
Grid points visited: 3792099  (42%)
Grid points evaluated: 3472439  (39%)
Global minimum: -22058 at x=-2165,3 y=-641,7
```

Increasing search density by 0.1 starting at 0.1 search density, keeping all other values constant



```
PS C:\Users\kent1\OneDrive - University of Cape Town\CSC2002S\Week 1\Assignment 1\AssUnix\ParallelAssignment2023(7) (1)>  c:;
cd 'c:\Users\kent1\OneDrive - University of Cape Town\CSC2002S\Week 1\Assignment 1\AssUnix\ParallelAssignment2023(7) (1)'; & '
C:\Program Files\Java\jdk-11.0.8\bin\java.exe' '-cp' 'C:\Users\kent1\OneDrive - University of Cape Town\CSC2002S\Week 1\Assign
ment 1\AssUnix\ParallelAssignment2023(7) (1)\bin' 'MonteCarloMini.MonteCarloMinimizationParallel' '1000' '1000' '-3500' '3500'
 '-3500' '3500' '0.1'
Run parameters
         Rows: 1000, Columns: 1000
         x: [-3500,000000, 3500,000000], y: [-3500,000000, 3500,000000]
         Search density: 0,100000 (100000 searches)
Time: 81 ms
Grid points visited: 177825  (18%)
Grid points evaluated: 333457  (33%)
Global minimum: -22058 at x=721,0 y=7,0

Rosenbrock function: 2.7022739274E13
Run parameters
         Rows: 1000, Columns: 1000
         x: [-3500,000000, 3500,000000], y: [-3500,000000, 3500,000000]
         Search density: 0,200000 (200000 searches)
Time: 48 ms
Grid points visited: 231309  (23%)
Grid points evaluated: 297840  (30%)
Global minimum: -22058 at x=595,0 y=-1666,0

Rosenbrock function: 1.2651609100936E13
Run parameters
         Rows: 1000, Columns: 1000
         x: [-3500,000000, 3500,000000], y: [-3500,000000, 3500,000000]
         Search density: 0,300000 (300000 searches)
Time: 64 ms
Grid points visited: 317097  (32%)
Grid points evaluated: 351870  (35%)
```

Report: Discussion:

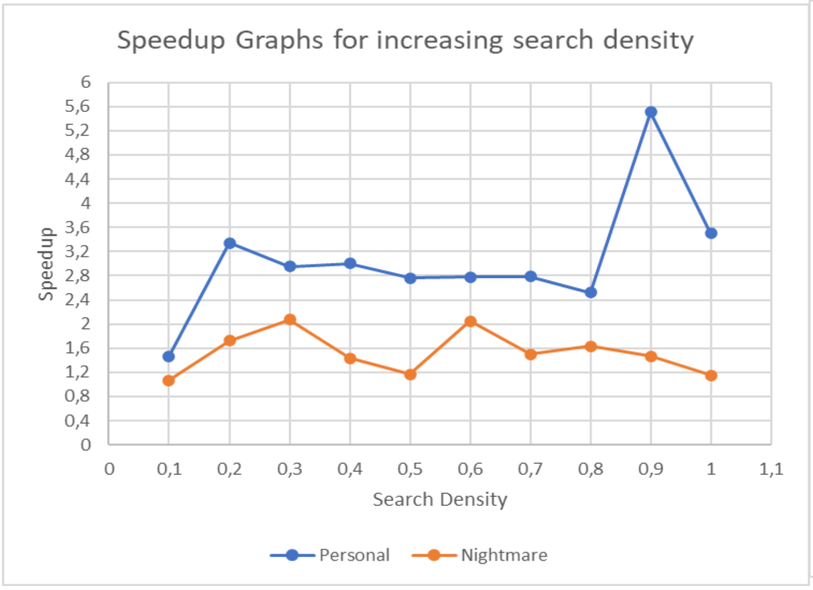
These are speedup graphs which I calculated using T1/Tp that I ran for 10 tests, the first side-by-side graphs represent the change in speedup of increasing search density by 0.1, one side is when I ran the program on the nightmare architecture and the other side is when I ran the program on the Personal computer architecture. Then I will compare to the ideal speedup graphs.

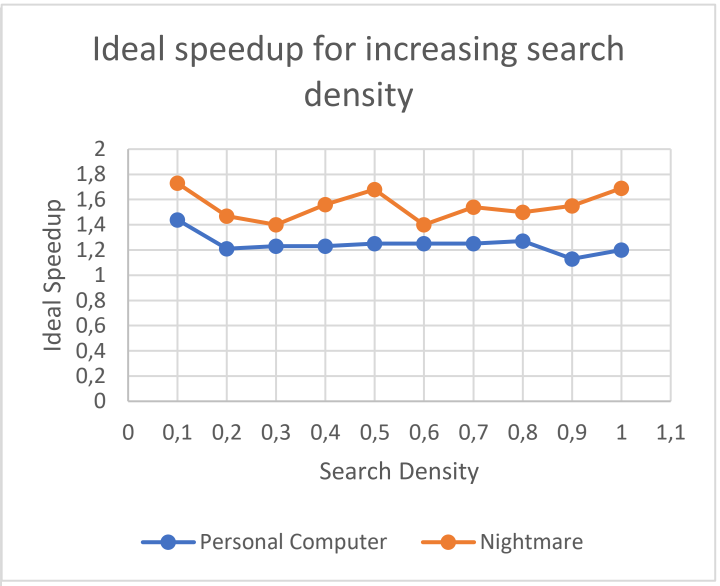
How I calculate ideal = example: in Nightmare (8 cores)

Parallel solution takes 17ms; Serial Solution takes 21ms

Normal = 21/17 = 1,24 speedup

Ideal = 1/ (1-17/38) + (17/38)/8 = 1,64 ideal speedup





As you can see, the actual speedup for the personal computer architecture performed consistently and more effectively than Nightmare. After the 0.1 density, it exhibited a noticeable spike and maintained a stable performance until a density of 0.9. Subsequently, it achieved its maximum speedup. However, at a density of 1, the speedup was lower due to the reduced number of searches, even though the architecture comprises only 4 cores. The speedup consistently surpassed both the ideal speedup and the maximum speedup across all densities. Notably, while the ideal speedup is generally lower than the Nightmare ideal speedup, the actual speedup still exceeded these benchmarks. This might explain the substantial difference in execution time between the serial program on Nightmare and the parallel program, with the latter achieving significantly lower execution times. The maximum speedup at 0.9 can be attributed to the processing of a substantial number of searches by the parallel threads, resulting in a considerably faster execution than sequential processing. The performance of nightmare was more consistent but none of the speedup numbers reaches the heights of personal computer architecture, even though it has 4 more cores than the personal computer architecture. It’s performance almost matches the overall consistent performance of the ideal speedup graphs, although slightly lower at some points. The overall lower speedups explain why both the general time in ms generated by the serial program was less than the general time in ms generated by the personal computer architectures, same with the time in ms generated by the parallel algorithm. 0.3 and 0.6 had the maximum speedup while in the ideal, it had the lowest speedup explains why more cores process both the serial and parallel faster than 4 cores, resulting in less speedup calculated.

Comparatively, Nightmare exhibited more consistent performance, although its speedup values did not match the heights reached by the personal computer architecture. Despite having 4 additional cores, the performance of Nightmare did not reach the same speedup levels. Interestingly, its performance closely resembled the overall consistent performance of the ideal speedup graphs, albeit slightly lower at some points. These lower speedup values explain the trend of the general execution time in milliseconds being lower in both the serial and parallel programs compared to the personal computer architecture. This discrepancy is evident for both the serial and parallel algorithm execution times. The densities of 0.3 and 0.6 demonstrated the maximum speedup in Nightmare, aligning with the observation that more cores led to faster processing in both the serial and parallel contexts. This, in turn, resulted in lower calculated speedup values

This pattern explains why the speedups did not reach the maximum values attained by the personal computer architectures. With 8 cores operating more efficiently for both serial and parallel processing, the calculated speedup inevitably remained lower than that of the 4-core setup.

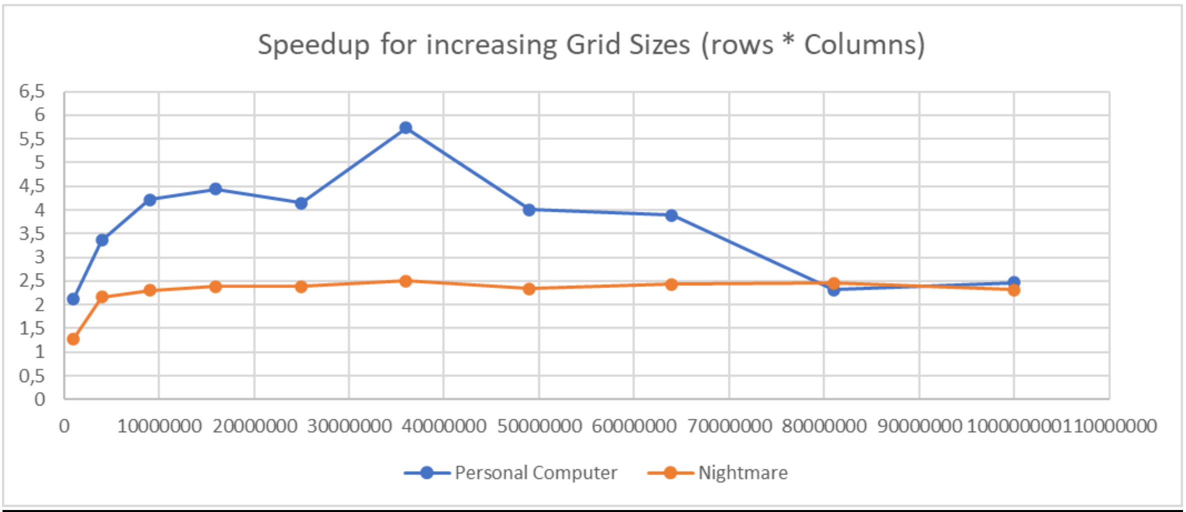
Trends and Anomalies:

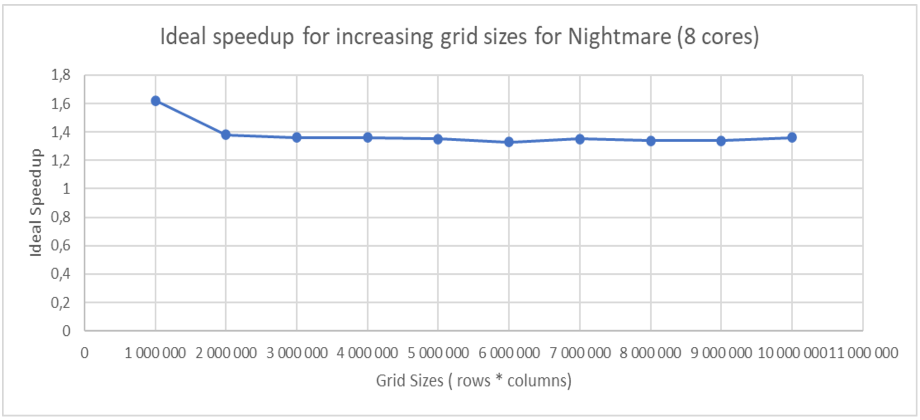
In Nightmare, densities of 0.3 and 0.6 yielded the maximum speedup, which aligns with the observation that more cores lead to faster processing in both serial and parallel contexts. Contrarily, in the ideal scenario, the lower speedup values can be attributed to the more efficient processing by additional cores, resulting in less significant speedup.

For the personal computer architecture, an interesting spike in the speedup occurred at a density of 0.9. Prior to this, the actual speedup experienced a steady decline as search density increased. However, at a search density of 1, the speedup was notably lower, despite a larger number of searches being processed. Aside from this anomaly, the speedup generally exhibited an increasing trend as search densities grew larger. Conversely, in the case of Nightmare, the speedup remained consistent, with a few instances of lower values.

Report discussion:

Increasing Grid Sizes (Rows * Columns):





The same applies to the grid sizes as did with the search density results, the general speedup of the personal computer was overall higher than the nightmare speedup processes, with the Personal computer architecture, it gradually increased its speedup until its spike into it’s maximum speedup with 5,73, then gradually increased, it seems the parallel algorithm lost its effectiveness after the middle of the graph where the rows and columns surpassed the number of 5000 and as the time in ms generated by the serial and parallel algorithm got closer and closer, since the calculated speedup got smaller. It seems after the grid size got too big, the threads were less effective in parallelising the process.

For Nightmare, it gradually increased from 1000 rows and columns and then for the entire graph, it stayed consistent, with its maximum reaching 2.5 speedup. The reason it stayed consistent could be the same reason why the 8 cores processed the serial and parallel just as fast. But as the grid size got larger, it still stayed consistent whereas the personal computer speedup deceased till it got to the same level as Nightmare.

The general speedup for the Nightmare architecture was overall higher than the ideal speedup. This pattern clarifies the reason why the speedups didn't achieve the highest values observed in the personal computer architectures. Due to the superior performance of 8 cores in handling both sequential and parallel tasks, the resulting calculated speedup naturally stayed below that of the 4-core configuration.

Trend and Anomalies:

After the spike of 5.73 in the personal Computer architecture, the speedup gradually decreased as grid sizes got bigger. For Nightmare, the speedup was consistent throughout the graph with no noticeable spikes.

Conclusion:

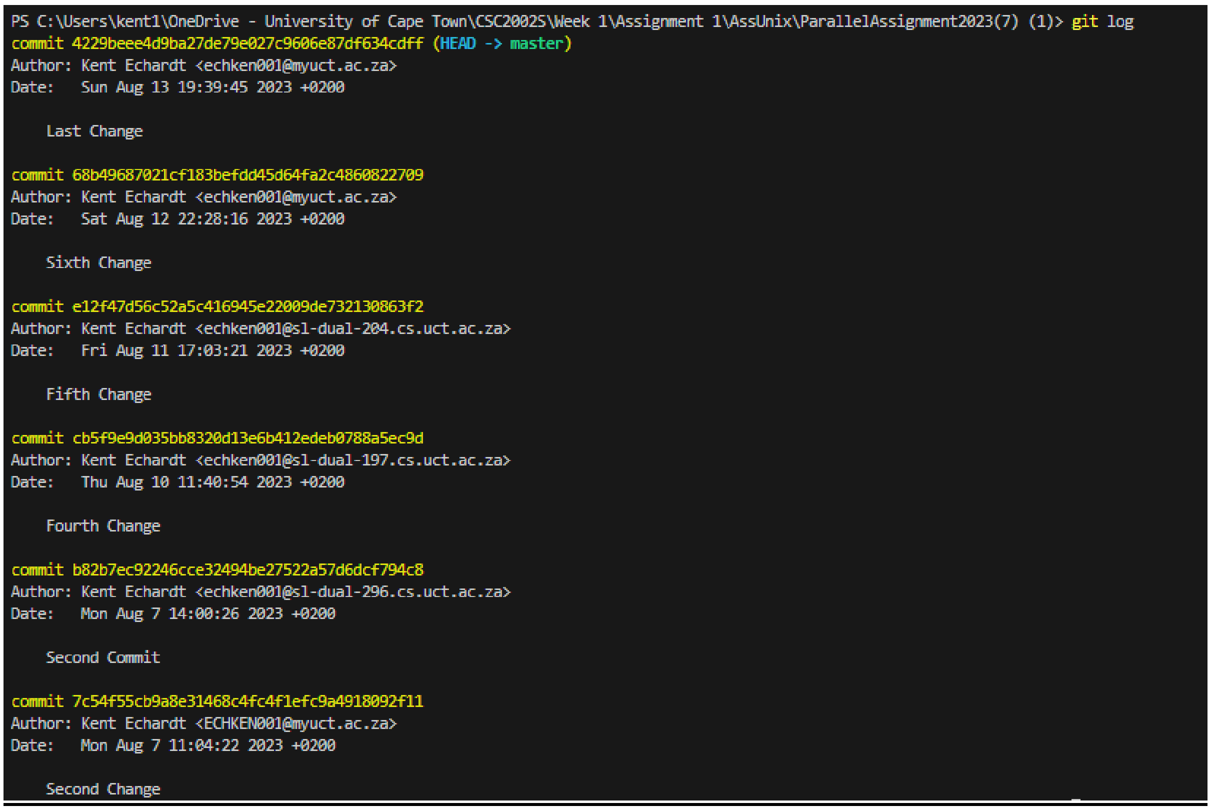
In conclusion, parallelising the Monte Carlo minimization algorithm did indeed yield significant insights and was worth getting beneficial results. Our goal was to leverage the computational power of multicore processors to enhance the efficiency of the algorithm, with a focus on achieving speedup and improved performance.

The results we got provided a clearer view of how of how parallelization works on the algorithm's execution. We saw how different the speedup results are with different search densities and grid sizes, in which search density has the most change due to the noticeable spikes in its speedup, while in the changing grid sizes, both the time in ms generated by the serial and parallel algorithms took just as long, resulting in lower and more consistent speedups.

The speedup achieved consistently surpassed the ideal speedup, even though the ideal speedup remained slightly lower in the context of the personal computer architecture's ideal speedup. Notably, the performance had a significant boost when increasing the number of threads up to a certain point, beyond which diminishing returns were observed.

Overall, the architecture with the least cores had the largest spikes and speedups while the architecture with the most cores rather stayed consistent with it’s speedups due to it’s processing power both affecting the time generated by the serial and parallel algorithm, resulting in the least speedups.

Git Log:



```
PS C:\Users\kent1\OneDrive - University of Cape Town\CSC2002S\Week 1\Assignment 1\AssUnix\ParallelAssignment2023(7) (1)> git log
commit 4229beee4d9ba27de79e027c9606e87df634cdff (HEAD -> master)
Author: Kent Echardt <echken001@myuct.ac.za>
Date:   Sun Aug 13 19:39:45 2023 +0200

    Last Change

commit 68b49687021cf183befdd45d64fa2c4860822709
Author: Kent Echardt <echken001@myuct.ac.za>
Date:   Sat Aug 12 22:28:16 2023 +0200

    Sixth Change

commit e12f47d56c52a5c416945e22009de732130863f2
Author: Kent Echardt <echken001@sl-dual-204.cs.uct.ac.za>
Date:   Fri Aug 11 17:03:21 2023 +0200

    Fifth Change

commit cb5f9e9d035bb8320d13e6b412edeb0788a5ec9d
Author: Kent Echardt <echken001@sl-dual-197.cs.uct.ac.za>
Date:   Thu Aug 10 11:40:54 2023 +0200

    Fourth Change

commit b82b7ec92246cce32494be27522a57d6dcf794c8
Author: Kent Echardt <echken001@sl-dual-296.cs.uct.ac.za>
Date:   Mon Aug 7 14:00:26 2023 +0200

    Second Commit

commit 7c54f55cb9a8e31468c4fc4f1efc9a4918092f11
Author: Kent Echardt <ECHKEN001@myuct.ac.za>
Date:   Mon Aug 7 11:04:22 2023 +0200

    Second Change
```