METHODS

Parallelization was achieved by using the fork-join framework designed for divide-and-conquer algorithms to perform multiple independent Monte Carlo searches in parallel. The MonteCarloMinimization.java program imports **ForkJoinPool** and the **RecursiveTask**, it then extends the **RecursiveTask<Integer>**.

The program initializes the necessary parameters such as the grid size, terrain limits and search density. It creates an instance of the TerrainArea class to represent the terrain and generate random searches. The program creates an array of '**Search**' objects to represent the individual Monte Carlo searches. Each search is assigned a random starting position on the terrain.

The **compute()** method of MonteCarloMinimization.java is invoked by the Fork-Join framework. This method splits the task into subtasks for parallel execution. If the difference between the **high index** and **low index** of the task at hand (number of searches) is less than or equal to the **'SEQUENTIAL_CUTOFF'**, each search is performed sequentially by iterating through the array of searches and finding the local minima. Otherwise, if the difference is bigger to handle or greater than the **'SEQUENTIAL_CUTOFF',** the task is split into two subtasks and shared to threads using recursion. The **fork()** method is used to initiate the execution of the left subtask. It schedules to run in a separate thread, allowing it to run concurrently with the current task. The right subtask calls the **compute()** method because the right subtask was forked and is running asynchronously, but the right subtask's computation is performed directly in the current thread. The **join()** method blocks the current thread until the result of the left subtask is available. Then, the minimum value from both subtasks is calculated by comparing the results of the left and right subtasks using the **Math.min()** function. The minimum is assigned to the variable 'min', which represents the global minimum found in the entire search space.

Finally, to start multithreading the MonteCarloMinimization and the ForkJoinPool objects are created. The **invoke()** method of the ForkJoinPool is used to give the task to the pool.

To optimize the program the **SEQUENTIAL_CUTOFF** was used. This is a base case were the program stops to run recursively. It is where the algorithm stops parallel execution i.e., giving tasks to worker threads and switches to sequential execution. This is important for optimization because as the problem size gets bigger the **SEQUENTIAL_CUTOFF** must increase to utilize the threads power to run more tasks in parallel otherwise if not utilized the computational power goes to waste. When **SEQUENTIAL_CUTOFF** is low, and the number problem is larger the parallel code takes long to compute.

To validate the algorithm efficiency the Rosenbrock function was used on the serial and parallel program to compare the outputs.

Serial output:

Run parameters

Rows: 2000, Columns: 2000

x: [-200.000000, 200.000000], y: [-200.000000, 200.000000]

Search density: 0.050000 (200000 searches)

Time: 58 ms

Grid points visited: 217341 (5%)

Grid points evaluated: 1198962 (30%)

Global minimum: 0 at x=1.0 y=1.0

Parallel output:

Run parameters

Rows: 2000, Columns: 2000

x: [-200.000000, 200.000000], y: [-200.000000, 200.000000]

Search density: 0.050000 (200000 searches)

Time: 68 ms

Grid points visited: 205045 (5%)

Grid points evaluated: 882770 (22%)

Global minimum: 0 at x=1.0 y=1.0

To benchmark the algorithm for the first part of the experiment the input data used was from **0 to 9750 (rows and columns**) with **scale of 750** apart each problem size. The number totalled to a range of **0 to 4 753 125 searches** by keeping the search density the same at 0,05. The ranges xmin, xmax, ymin, ymax were also kept the same. First the inputs were put in the makefile in this format **750 750 -200 200 -200 0,05** then it was run a couple times for the same input to figure out the lowest time taken to find the global minimum. The outputs of each problem size were recorded. This was done on both the laptop and the department server for both the serial and the parallel program.

Then the times for each problem sizes were taken to calculate the speed-up value between the serial and the parallel program. For each problem size time of serial was divided by time of parallel to obtain the speed-up. These speed-ups were used to draw trendlines for both the laptop and departmental server. The speed-up graph is titled **Speed-up vs. Problem size graphs.**

The second part used varied number of search densities and kept other variables constant. Rows and columns were given a random number 4500 for each and the ranges xmin, xmax, ymin, ymax were kept the same. The range of densities used ranged from 0,1 to 0,9 with steps of 0,1. These inputs were tested on my laptop and the department using both the serial and parallel program while recording the times for each algorithm. Then again, these times for different search densities were used to find the speed-up value for each search density. The speed-up values were used to draw the **Speed-up vs Search density graphs.**

The third part of the experiment used very large rows and columns ranging from 10000 to 30000 with a scale of 2500. Other variables were kept constant. These inputs were only run on the department server. The outputs for each problem size of both the serial and parallel program were taken to calculate the speed-up value. The time for serial program was divided by that of a parallel program.

See specifications below of the architecture used to conduct the experiment:

1.Laptop

Architecture: x86_64

CPU op-mode(s): 32-bit, 64-bit

Byte Order: Little Endian

CPU(s): 4

On-line CPU(s) list: 0-3

Thread(s) per core: 2

Core(s) per socket: 2

Socket(s): 1

Vendor ID: GenuineIntel

CPU family: 6

Model: 142

Model name: Intel(R) Core(TM) i3-8145U CPU @ 2.10GHz

Stepping: 12

CPU MHz: 2304.008

BogoMIPS: 4608.01

L1d cache: 32K

L1i cache: 32K

L2 cache: 256K

L3 cache: 4096K

2.Department server

Architecture: x86_64

CPU op-mode(s): 32-bit, 64-bit

Address sizes: 40 bits physical, 48 bits virtual

Byte Order: Little Endian

CPU(s): 8

On-line CPU(s) list: 0-7

Vendor ID: GenuineIntel

Model name: Intel(R) Xeon(R) CPU E5620 @ 2.40GHz

CPU family: 6

Model: 44

Thread(s) per core: 2

Core(s) per socket: 4

Socket(s): 1

Stepping: 2

Frequency boost: enabled

CPU max MHz: 2401.0000

CPU min MHz: 1600.0000

BogoMIPS: 4788.37

Caches (sum of all):

L1d: 128 KiB (4 instances)

L1i: 128 KiB (4 instances)

L2: 1 MiB (4 instances)

L3: 12 MiB (1 instance)

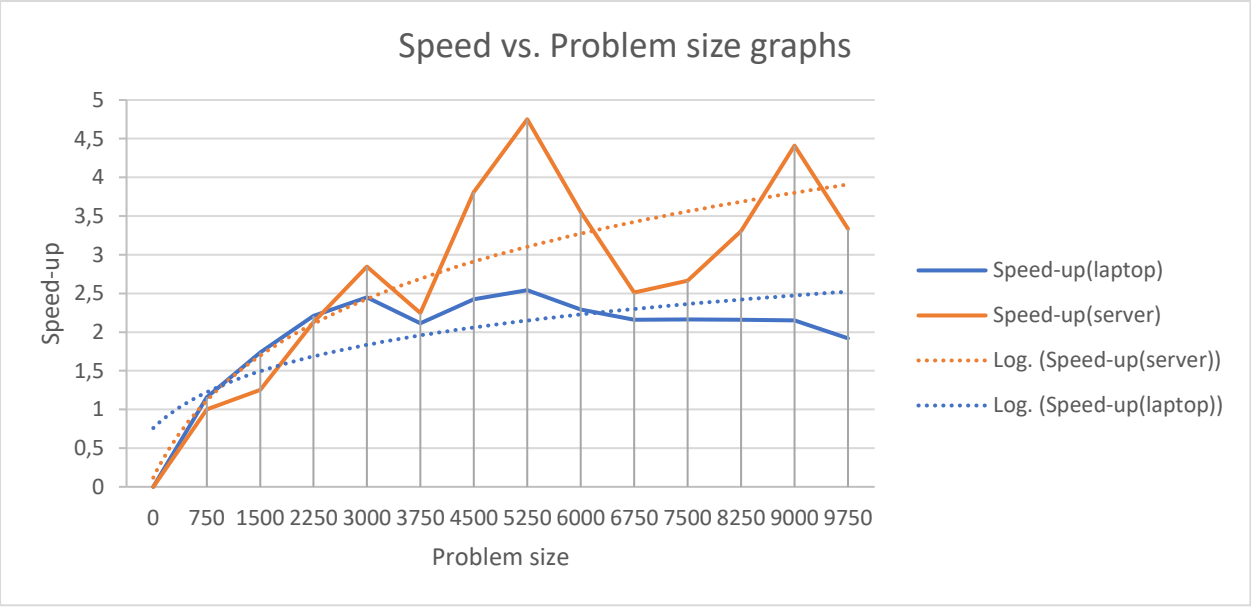
Some of the problems that were encountered during the process was that at first I had changed the Search.java program **find_valleys()** method to compute() to make use of recursion. The problem was that this method does not have any tasks to complete, it just lands somewhere random on the surfaces and then moves downhill, stopping at the local minimum to find the local minimum. Furthermore, the **find_valleys()** could not be changed in such a way that could use the **fork()**, **join()** and **compute()** methods of the fork-join framework it only used the **invoke()** function therefore the program could not get any faster than the serial code. The part that is most crucial is the amount of time it takes to complete the number of searches to find the global minimum.

Other problem faced was with java heap space memory because my machine uses the Windows Subsystem for Linux(WSL) to run the parallel program. The error that I kept on receiving was '**java.lang.OutOfMemoryError: Java heap space'** which indicates that the Java program is running out of memory. This happens when the program needs to allocate more memory than is available in the Java heap. This often happened when I tried to put extremely large grid sizes. The departmental server also gave the same problem but not as bad as my machine. My laptop JVM memory max heap size is 940 MB while the department server is 12 GB. My laptop could only handle less than 5 000 000 searches and the departmental server approximately less than 45 000 000 searches. This problem restricted the flexibility of the experiment. I could only use very large values on the server.

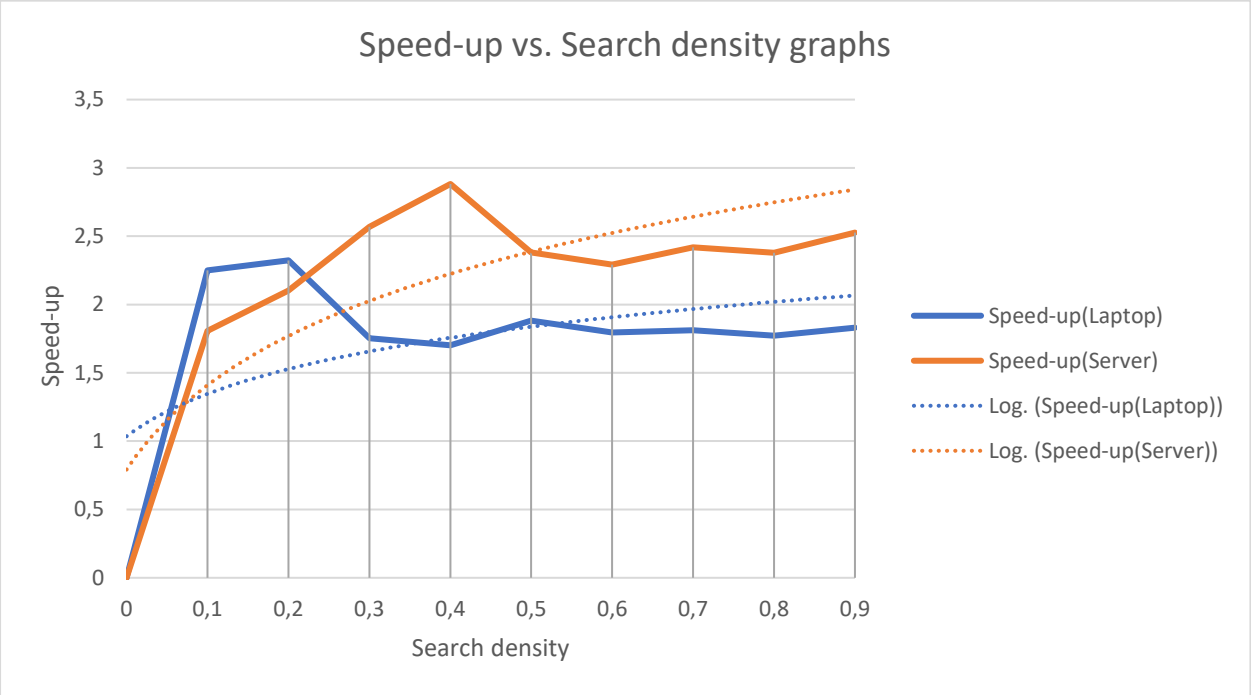
RESULTS

In this graph below it can be seen that parallel program performs better for values greater than 2250, this is the point where the parallel program starts to perform 2x more than the serial program. Because it is at that point were the program performs more searches where the workload is balanced, and worker threads use their computational power. Furthermore, it can be observed that at first the laptop was performing better than the department server(nightmare) when the problem size was less than 2250. Then, when the problem size starts to get larger at 3000 and going upwards the department server trendline starts to surpass that of the laptop with high margins; the highest speed up was close to 5x speed. While the laptops trendline remained constant but was still above 2x speed up.

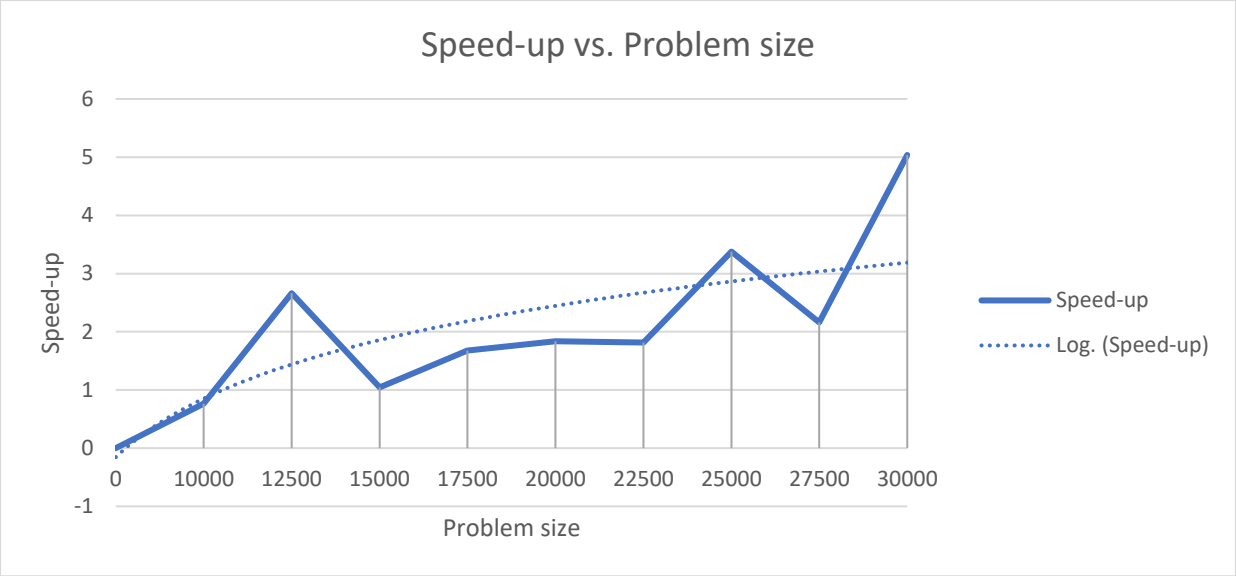
On the server speed-up graphs it can be observed that there are large spikes. The speed-up went up to highs of close to 5x speed and dropped drastically to almost 2,5x and started to go up again to almost 4,5x speed. The impact of this might have been caused by the race condition in case where threads write simultaneously to the same grid location, this however does not impact the global minimum but does have a slight impact on the time taken to find the global minimum.



In this graph below it can be observed that when the search density is less than 0,2 the laptop performs better than the department server. Then, when the search density is greater than 0,2 the department server graph surpasses that of the laptop. This is where the multicore machine takes uses its computational power. The department server has 4 cores per socket while the laptop has only 2 cores per socket and they both allocate 2 threads per core (see specifications above). Both the graphs remain constant after the 0,5 search density. Furthermore, there are spikes that can observed, at the beginning of the graph there's already a large spike at point 0,1. This might have been caused by using a very large grid size (4500 x 4500). There are too drops which happen on both graphs at different search densities. On the laptop's speed-up graph it happens when after 0,2 search density while on the department's server it happens after 0,4. This further shows that the multicore machine can handle large data before it drops to its constant level (optimal point) or diminishing returns.



This graph below shows the trendline of the speed-up graph using large problem sizes to see the impact it will on the speed of the parallel versus the serial program. The performance of a parallel performs better than the serial as the problem size increases; 5x speed is obtained when input of rows and columns is 30000 x 30000.



CONCLUSIONS

Even though the parallel program when tested on each architecture did not reach the ideal speed-up of a parallelization it was still worth it because all the graphs clearly show that with increasing problem size the speed-up also increases. The graphs also show that it is encouraged to use a parallel program when dealing with big values (or complex problem) because with small input sizes the speed-up was 1 or less. The graphs also show that with more computational power (multiple cores) the parallel program performs even more better. Also, the data shows it is important to validate and optimize a parallel program by using the **SEQUENTIAL_CUTOFF**. Which brings another conclusion that the implementation of parallelization must be done in a correct and most efficient way to obtain viable results.