PCP Assignment 1

Paralleziation approach

My approach to parallelizing the Montecarlo minimisation algorithm is using the fork-join framework. The fork-join framework is designed to speed up the execution of tasks that can be divided into other smaller subtasks, executing them in parallel and combining their results to get a single one.

To perform the parallel computations I made a class called Grid Searcher that will be responsible for doing the task. The GridSearcher class extends RecursiveTask<Integer[]>, which is a part of the Fork/Join framework. This class is used to perform a computation that might be split into smaller computations that can be performed in parallel. The compute() method is overridden to provide the computation that needs to be performed.

The GridSearcher class is initialized with an array of Search objects and the number of searches to be performed. It also maintains a localMin variable to keep track of the minimum value found in the current task, and a min variable to keep track of the overall minimum value found across all tasks.

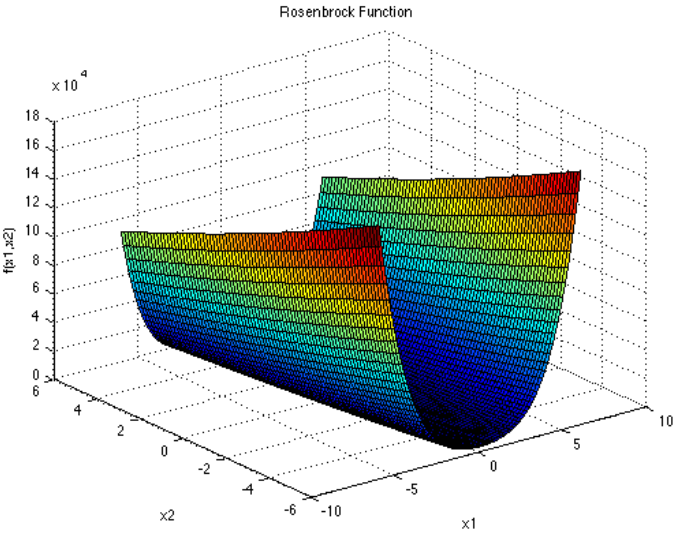
In the compute() method, if the number of searches is less than a threshold, the method iterates over the Search objects and calls the find_valleys() method on each one. If the returned value is less than the current minimum, it updates the minimum and the index of the Search object that found it.

If the number of searches is greater than the threshold, the method splits the array of Search objects into two and creates two new GridSearcher tasks. It then invokes these tasks using the invokeAll() method, which executes them in parallel. After the tasks are complete, it compares the minimum values found by each task and updates the overall minimum and the index of the Search object that found it.

The MonteCarloMinimization class is the main class that creates a ForkJoinPool and submits the GridSearcher task to it. This approach balances the overhead of task creation and management with the benefits of parallel execution. The code also finds the minimum value from the tasks by keeping track of the minimum value found by each task and comparing them after all tasks have been completed. This approach ensures that the minimum value is found even when the task is divided into subtasks and executed in parallel.

Validation

Using the Rosenbrock function to generate to terrain and then performing a search on it, I validate my optimization algorithm. The Rosenbrock function provides a challenging terrain for the search algorithm, and it should find the global minimum which is zero as illustrated in the picture on the left that shows the Rodsebronk function and with that, the algorithm is considered working.



As shown below my algorithm is finding the global minimum zero when using the Rosenbrock function for different values of rows and columns and search densities. If x = a then y = a^2;

```
Run parameters
        Rows: 1000, Columns: 1000
        x: [-500.000000, 500.000000], y: [-500.000000, 500.000000]
        Search density: 0.500000 (500000 searches)
Time: 552 ms
Grid points visited: 394942  (39%)
Grid points evaluated: 2353309  (235%)
Global minimum: 0 at x=1.0 y=1.0

Run parameters
        Rows: 2000, Columns: 2000
        x: [-1000.000000, 1000.000000], y: [-1000.000000, 1000.000000]
        Search density: 0.300000 (1200000 searches)
Time: 1386 ms
Grid points visited: 1039474  (26%)
Grid points evaluated: 6218075  (155%)
Global minimum: 0 at x=1.0 y=1.0

Run parameters
        Rows: 3500, Columns: 3500
        x: [-1750.000000, 1750.000000], y: [-1750.000000, 1750.000000]
        Search density: 0.500000 (6125000 searches)
Time: 3977 ms
Grid points visited: 4798123  (39%)
Grid points evaluated: 23484992  (192%)
Global minimum: 0 at x=1.0 y=1.0

Run parameters
        Rows: 6000, Columns: 6000
        x: [-3000.000000, 3000.000000], y: [-3000.000000, 3000.000000]
        Search density: 0.050000 (1800000 searches)
Time: 2474 ms
Grid points visited: 1760754  ( 5%)
Grid points evaluated: 10545761  (29%)
Global minimum: 0 at x=1.0 y=1.0

Run parameters
        Rows: 10000, Columns: 10000
        x: [-5000.000000, 5000.000000], y: [-5000.000000, 5000.000000]
        Search density: 0.050000 (5000000 searches)
Time: 14531 ms
Grid points visited: 4884596  ( 5%)
Grid points evaluated: 29280013  (29%)
Global minimum: 0 at x=1.0 y=1.0

Run parameters
        Rows: 13000, Columns: 13000
        x: [-6500.000000, 6500.000000], y: [-6500.000000, 6500.000000]
        Search density: 0.010000 (1690000 searches)
Time: 4188 ms
Grid points visited: 1686508  ( 1%)
Grid points evaluated: 10103377  ( 6%)
Global minimum: 0 at x=1.0 y=1.0
```

Benchmarking process

In the MonteCarloMinimization class, I implemented this by capturing the system time before and after the execution of the algorithm.

Using the tick() method, I captured the start of the execution and the tock() method to capture the system time at the end of the execution. Using the difference between the end time and the start time I am able to compare the performance of different implementations or optimizations.

Machine architectures

My local machine

OS name: Microsoft Windows 11 Home single language
System details: Acer Aspire A315-57G
Processor: Intel(R) Cre(™) i5-1035G1 CPU @ 1.00GHz, 1190 <hz, 4Core(s), 8 Logical cores

Remote machine: UCT Nightmare server

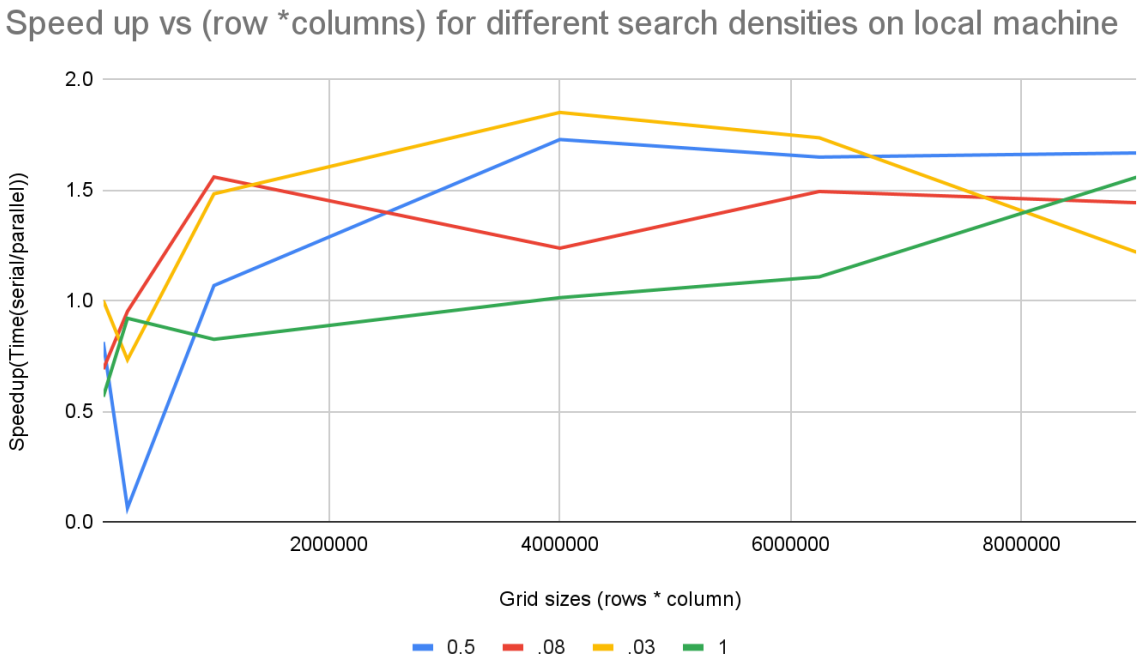
OS name: Ubuntu 2,04,2 LTS
Processor: Intel(R) Xeon(R) CPU E5620 @ 2.40GHz, MHz, 4 Core(s), 2 Logical cores
CPU family:6
Model: 44
Thread(s) per core: 2
Core(s) per socket:4

Issues and problem

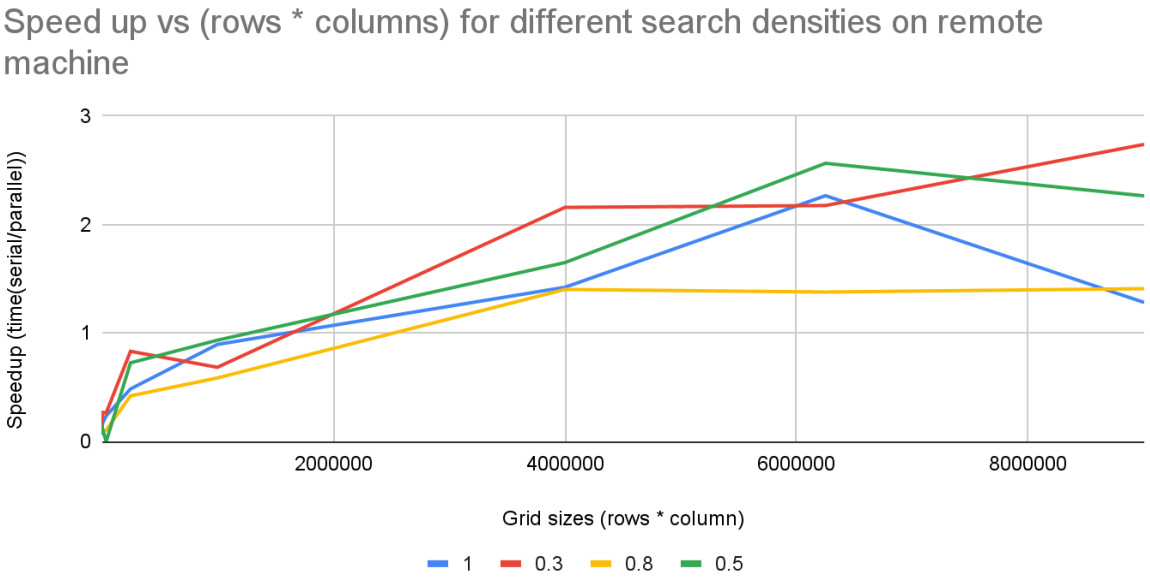
1. Java Heap space running out: When dealing with high grid sizes with large densities I encountered memory issues. My programs tried to use more memory than is available in the heap space.
2. Concurrency issues: I encountered race conditions where the threads wrote simultaneously to the same grid location.
3. Testing and debugging: Testing and debugging parallel code is more complex than sequential code.
4. Performance tuning: Task creation and management when trying out different implementations while maintaining optimal performance is complex and time-consuming.
5. Hardware limitation: due to the code being heavily dependent on the local hardware I was running, I wasn't able to get the expected performance for big grid sizes with high densities.

Results

On local machine



On remote machine



Discussion on the results

The following graphs measure the relationship between the speedup and grid size for different search densities on the local and remote machines.

For both architectures, the speedup generally increases as the grid size increases, indicating that parallel execution becomes more beneficial for larger problem sizes.

The speedup is not proportional to the grid size. For example, in the local architecture with a search density of 0.5, the speedup increases from 0.061 (grid size of 100) to 1.649 (grid size of 2000). However, the speedup drops to 2.264 for a grid size of 2500, which is unexpected and requires further investigation.

The speedup varies across different search densities. For instance, in the local architecture, the speedup with a search density of 0.3 is generally lower than the speedup with search densities of 0.5 and 0.8. This suggests that the computational workload and parallelizability of the algorithm vary with the search density.

Finding the ideal maximum speedup was not possible as I could not temper the number of professors in the fork-join pool.

Conclusion

This is expected behaviour when parallelizing an algorithm. The overhead of managing multiple threads can make the parallel version slower for small inputs, but as the input size increases, the benefits of parallel execution start to outweigh the overhead, leading to better performance.

In terms of the percentage of grid points visited and evaluated, both versions seem to perform similarly. This indicates that the parallel version is correctly implementing the same algorithm as the sequential version.

In conclusion, parallelization was worth it for larger grid sizes, as it led to significant performance improvements. However, for smaller grid sizes, the performance was similar to the sequential version. This suggests that a hybrid approach might be the most efficient solution, where the sequential version is used for small inputs and the parallel version is used for large inputs.