CSC2002S Assignment PCP1 2023 Report

Parallelizing Monte Carlo Function Optimisation

Qailah Bhamjee (BHMQAI001)

Methods

Parallelization Approach and Optimisations

The parallelization approach I utilised is based on the Fork/Join framework, which uses a divide-and-conquer strategy to share tasks among multiple threads.

The divide-and-conquer strategy breaks down the 2D grid by halving the rows of the grid multiple times, so that each division can be processed concurrently. The class MonteCarloParallel extends RecursiveTask<Integer>, which is part of the Java Fork/Join framework, and represents a task that can be divided into smaller subtasks and then computed in parallel. The threshold value determines the size of these subtasks. If the range of rows to be processed is below the threshold, the compute method performs searches for the minimum value sequentially. If the range of rows is above the threshold, the compute method performs parallel execution. This involves dividing the range into half and creating two new subtasks to each process one of the halves. The subtasks are then further divided using the fork method. Afterwards, the compute method uses the join method to retrieve the results of each subtask, which are then combined to determine the overall minimum value.

In my main method, I created an instance of MonteCarloParallelization and an instance of ForkJoinPool to manage the parallel execution. The invoke method of ForkJoinPool handles the distribution of tasks among threads and returns the overall result.

By setting a threshold value, the program is optimised by avoiding excessive task splitting for small subtasks. Initialising a TerrainArea instance before parallelization starts avoids redundant object creation with the parallel tasks, improving memory efficiency. Utilising the join method to wait for subtasks to compute further optimises the program by ensuring effective combinations of the parallel outcomes.

This approach allows for effective utilisation of multiple threads, as well as optimized performance.

Validation of Algorithms

I validated my parallel algorithm by comparing the results of my parallel program to the serial program for various grid sizes and search densities, to ensure that the global minimums are the same for the same input parameters. I ensured that the grid points visited and evaluated for the parallel program were less than that for the serial program, as well as the number of searches.

The Rosenbrock function (which was included in the code of TerrainArea) was used to further validate my parallel program.

Algorithm Benchmarks

I benchmarked my algorithm by timing its execution on two different machines and varying the grid size and search density.

By timing the execution of the program on two different machines, I could test that my algorithm runs on machines with different cores and investigate the effect of cores on execution time. I used my current laptop (Acer Swift SF314-59) which has 8 cores as one of my machines, and an old laptop (Dell Inspiron 3520) which has 4 cores as the other machine.

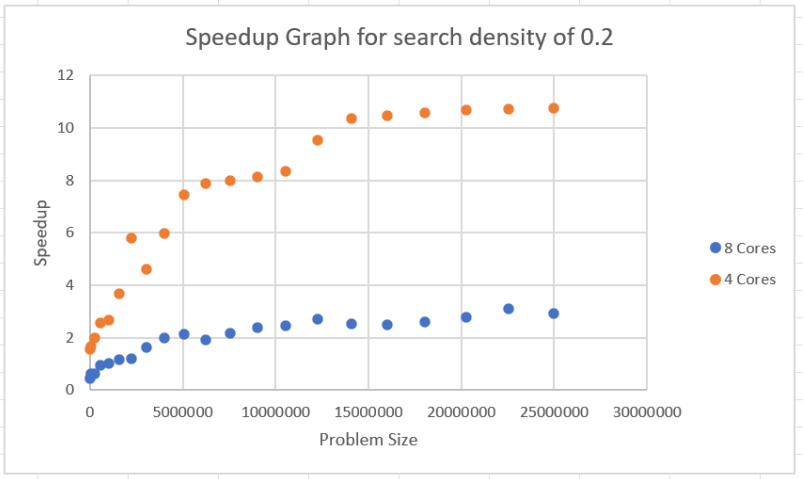
Varying the grid size and search density allowed me to create speedup graphs, which visually show how parallel algorithms scale. I varied the grid size from 100x100 to 5000x5000, and measured with search densities of 0.25, 0.5 and 0.75.

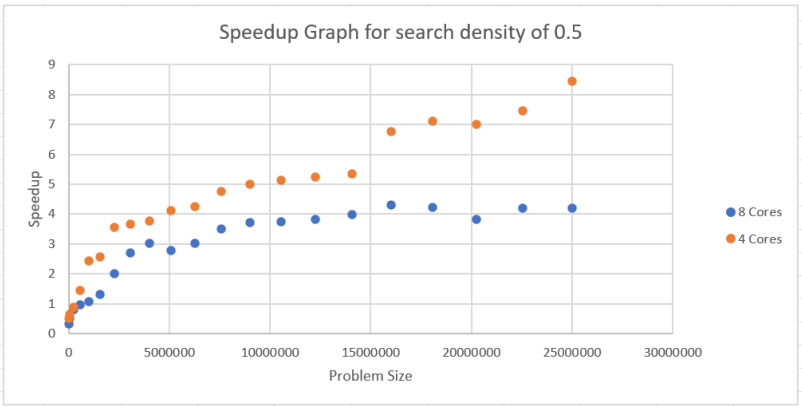
Problems/Difficulties Encountered

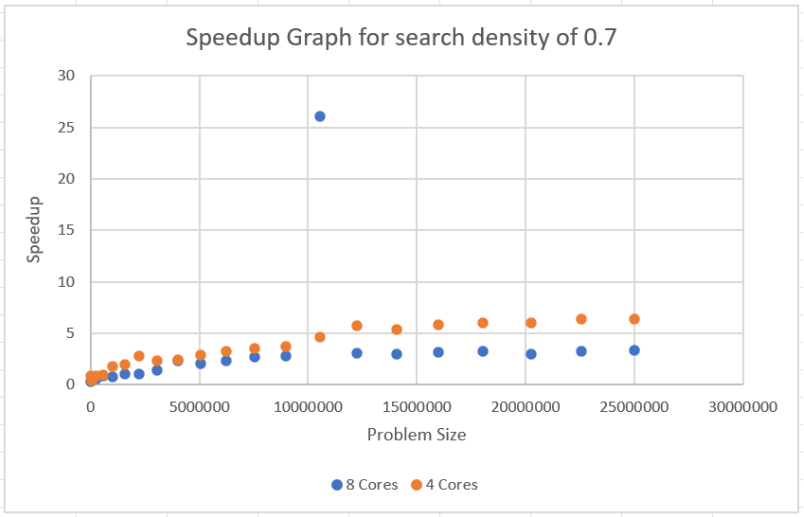
I encountered an issue where the global minimums from the parallel algorithm were not the same as those of the serial algorithm. The values were very close to the serial value, but not exact. This issue was dealt with by correctly implementing the TerrainArea class and creating a next_steps class specific to the SearchParallel class.

Results

Speedup Graphs







Discussion

My parallel program performed well for grid sizes of range 1250x1250 and up, due to the threshold value being set to 100. For larger problem sizes, my parallel algorithm was faster and more efficient.

The ideal speedup is equal to the number of cores of my machine. When testing on the Dell, the maximum speedup obtained was 10.5 , which is far from the ideal speedup of 4. When testing my algorithm on the Acer, the maximum speedup obtained was 4.43 , which is roughly half of the ideal speedup expected of 8.

My results are within the range that I expected, hence I consider them reliable. The speedup graphs obtained are similar in trend to that of parallel algorithms, as well as to the example graphs we were shown in class.

There are an anomaly in the speedup graph with search density 0,7. I restarted my machine due to some network problems, and the spike coincides with the first run of the program immediately after I started my machine back up, which may have caused the execution time to be so high.

Conclusions

Parallelization and multithreading can provide significant benefits when dealing with this problem in Java.

Since this problem allows for a large computational workload with a substantial grid size, it is more effective to use parallelization as the workload can be distributed among multiple threads.

This problem can also be very easily and effectively parallelized, as each search is independent of others and thus the serial algorithm is almost inherently parallel already. The multiple independent simulations that can be performed concurrently.

Since both the machines I tested on have multicore processors and efficient processing power, parallelization is more beneficial.

In conclusion, parallelization is worth in the case of the Monte Carlo minimization problem. The independent nature of simulations allows efficient use of multiple CPU cores, leading to significant performance improvements on multicore systems.