**CSC2002S Assignment PCP1 2023**

**VLNSHA004**

**Parallel Programming with Java:**

**Parallelizing Monte Carlo Function Optimisation**

**Report:**

**Introduction:**

An experiment was conducted to determine the conditions under which parallelizing a serial solution is worth the extra effort. The serial solution in question was a Monte Carlo method used to find the minimum(lowest) point of a two-dimensional mathematical function within a specified range. The given area is represented as a grid of points. Due to high computational costs, the Monte Carlo algorithm probabilistically searches for the function's minimum without evaluating all points. It uses a series of searches to achieve this.   
Each search is independent of other searches aside from the fact that all searches share one terrain area.

The experiment required data collection benchmarked on 2 machines namely a 10 core, 12 logical processor machine and a remote connection to the University of Cape Town Nightmare server, which has about 8 logical processors with 4 cores. The time taken for processing the serial and parallel solutions were compared on each machine to determine when the speedup achieved by a parallel solution to the Monte Carlo optimization algorithm.

**Method:**

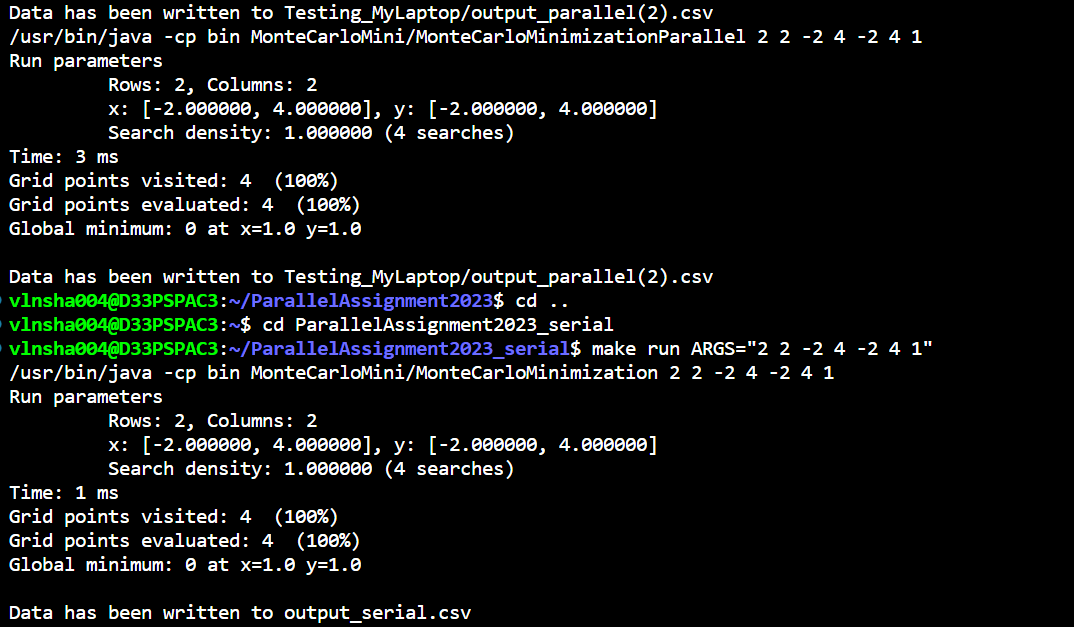
Parallelization approach:

To further elaborate on the nature of this experiment, the parallelization approach consisted of splitting the number of searches among threads, where each thread got a portion of the array of search objects to work with. This is different to the serial approach where all searches were done sequentially one by one, and no searches were carried out at the same time. The MonteCarloMinimizationParallel class was extended as a RecursiveAction that would update the global minimum as any searches found a local minimum that was lower than the global minimum. The compute method in MonteCarloMinimizationParallel contains code that would recurse through an array of searches and find the local minimum for each search. Optimizing the sequential cutoff proved to be a challenge, where a variety of sequential cutoffs were tested, including 500, 600 1000, 5000 and a function defined by SEQ\_CUTOFF = num\_searches\*0.1 or SEQCUTOFF = num\_searches\*0.05. It was found that a constant Sequential cutoff of around 500 was optimal for the speed of the program. The parallel programs were tested using a sequential cutoff of 500 for each program.

Validation approach:

The algorithm was validated using the Rosenbrock function that was compared between the serial and parallel version for a set of x and y values. Secondly, the minimum found by both the serial and parallel version was compared so that parallelizing the solution does not skew the accuracy of the solution. Both the serial and parallel version had to find the same global minimum in each program for the same set of parameters.

For example:



Both the parallel and serial solutions return the same global minimum when the Rosenbrock function is used to calculate the heights instead of the trig function defined in the given serial solution.

A computer screen shot of a black screen

Description automatically generated

Both the serial and parallel solutions return the same global minimum at the same co-ordinates.  
  
Benchmarking the algorithm:

The portion of the program that was timed was the searches where each search would find a local minimum and update the global minimum as the parallel program runs. This was timed on 2 different machines and the serial and parallel running times of the algorithm were compared on each machine. The 2 machines namely being a laptop with 10 cores (12 logical processors) and the departmental server(nightmare) with 4 cores(8 logical processors). Different grid sizes and search densities were tested. Furthermore, when timing the program, in order to get an accurate measure of the true speed of the algorithm, the program was run 4 times with the same set of parameters and the median time of the 4 times was used in generating the graphs in the results section. These median times were compared between serial and parallel solutions one each device.

Machine architectures:

As stated above, 2 machines were used to complete the experiment.

* Machine 1: Dell Inspiron 14 7420 with 10 cores and 12 logical processors.
* Machine 2: The nightmare departmental server with 4 cores and 2 threads per core. Hence, 8 logical processors.

The way the program was run on Machine 1 was on the Windows Subsystem for Linux, the Makefile was compiled and run through the terminal and timed in this way. Similarly, the sequential version of the program was timed in this fashion.   
On the nightmare departmental server, the connection was established on the windows subsystem for Linux and both versions of the program were run on the nightmare server.

There were slight difficulties or problems in timing the program using the WSL (Windows subsystem for Linux) where the results yielded little to no speedup from the serial to the parallel version. The speedup was compared between the 2 machines for the same parameters and on certain trials the nightmare server (Machine 2) was significantly faster than machine 1. It is worth noting the nature of the connection to the nightmare server: WSL was used to access a Linux terminal, whereafter a login to nightmare was successful through the terminal. This issue is elaborated upon in the results section.

**Results:**

Speedup plotted against grid size on Machine 1:

Speedup plotted against grid size on Machine 2:

The search density was varied between programs tested thus having sufficient c of grid sizes.

**Discussion:**

**Optimal Grid Size Range for Parallel Program Performance:**

**2 different behaviours on 2 different machines:**

In benchmarking the algorithm on machine 1(12 processors), it was found that the parallel Monte Carlo algorithm performs better as the size of the grid increases, implying a positive relationship between speedup and grid size, however after a certain point the speedup graph starts to decline, implying that the optimal grid size would be around 15,000,000. After this point we see a sharp decline on machine 1. On machine 2 the behaviour is different in that the optimal performance is for much smaller grid sizes of 400,000. There is still speedup thereafter, however it Is significantly lower speedup after a grid size of 400 000.

**Maximum Speed up and Comparison to Ideal Expected Speedup:**

Amdahl’s law defines the ideal expected speedup of a part serial, part parallel algorithm.



Simplifying Amdahl’s with S = 0 as this algorithm is embarrassingly parallel, we get a speedup of P where P is the number of Processors on the machine, this is equal to perfect linear speedup for a parallel algorithm.

The maximum speed up on machine 1 (12 logical processors) was around 1,471906 for a grid with 4000 rows and 4000 columns with a search density of 1. In this case the serial time was 4532ms and the ideal time of the parallel algorithm would be T1/P = 4532/12 = 377.67ms. If the parallel algorithm ran for 377.67ms, the ideal expected speedup of 12 would have been obtained. The highest speedup achieved however fell short of the bound of the ideal expected speedup.

The maximum speedup on machine 2 was 7,24 on the grid size 400 000. This alludes to a possible effect of not running the parallel program directly on a machine but instead a remote connection to a server.

**Reliability of Measurements:**

Measurements were taken by timing the program over 4 trials for the same set of parameters each time and taking a median of the four times the program ran. This was done on both machine 1 and machine 2, and for both the serial and parallel programs on both machines. Hence, the accuracy of the measurements is less skewed as the highest and lowest outliers of the trials do not skew the time the program took.

**Anomalies and Explanations:**

The sharp decline after a point for machine 1 and 2 suggests that management of threads and overheads plays a large role in affecting the performance of a parallel program. The longer a program takes to run and the more tasks it has to do at the same time, means the java fork join framework needs to spend additional time communicating with the processors and managing the manner in which tasks are processed. Hence why after a grid size of 16,000,000 on machine 1 the speedup started to decline as there were more processes to manage the longer the program ran. This combined with the need for the programs to communicate with the compatibility layer cause more overhead costs and more time is spent having to manage threads.

**Conclusion:**

In summary, the experiment aimed to assess the benefits of parallelizing a serial Monte Carlo method for function optimization. The results showed that the optimal grid size for parallelization varied between different machine architectures. While parallelization provided speedup, achieving perfect linear scalability proved challenging due to factors like thread management complexity and compatibility layers. The reliability of measurements was maintained through multiple trials. Anomalies in speedup curves were attributed to overhead costs and thread management intricacies. Overall, the experiment emphasized the nuanced nature of parallelization, where benefits are context-dependent and require careful consideration of machine setup, grid size, and overheads.