Montecarlo Minimization

Assignment 1

07/08/2023
CSC2002S

Nkhumeleni Best
NKHBES001

Table of Contents

Abstract

In the rapidly evolving landscape of computational techniques, parallelism has emerged as a powerful approach to enhance the efficiency and speed of various algorithms. This report presents the parallelization of the "Montecarlo Minimization" program.

Through a series of experiments and performance evaluations, we demonstrate the superiority of the "Montecarlo Minimization" program's parallelized approach over its linearized counterpart. The results exhibit not only remarkable speedup in execution time but also emphasize the potential for substantial gains in efficiency when dealing with computationally intensive optimization tasks.

Introduction

The Monte Carlo approach is a computational technique that relies on random sampling to estimate numerical results. While it's been a topic of debate, it can prove remarkably effective when dealing with systems or operations where our knowledge is limited.

In this report, we will employ the Monte Carlo approach to determine the lowest value within a defined region (terrain), characterized by a known mathematical function. Our goal is to accomplish this in two distinct ways, subsequently comparing the correctness and speed of each approach.

The first approach involves using a linearized version of the Monte Carlo method. In this scenario, we create a list of searches and systematically proceed through them, executing each search one after the other.

The second approach adopts a parallel strategy. Here, we assign each search to a separate thread, enabling multiple searches to be executed concurrently. This parallel execution holds the potential to expedite the process.

Methodology

1. ## Monte Carlo Approach Overview:

   The Monte Carlo approach essentially begins by selecting a random starting point. It then examines the values of nearby points in relation to this initial point. This process is particularly useful in our context, as we aim to identify the smallest value within the defined terrain.

2. ## Linearized Monte Carlo Approach:

   In the linearized version of the Monte Carlo approach, we start by crafting a list of searches. Each search is given a randomly selected row and column. The purpose of this process is to locate a valley, which signifies the nearest smaller value. As we detect these values, we store them in a designated variable. This variable essentially acts as a shared storage space to hold the smallest value we've encountered so far.

   As we proceed, the approach involves inspecting each search entry in the array sequentially. If a given search uncovers a value smaller than what we currently have in the shared variable, we update the variable to hold this new, smaller value. Conversely, if the search doesn't yield a smaller value, we simply move on to the next search entry and repeat the comparison process.

3. ## Parallelized Monte Carlo Approach:

   In my parallel implementation, I start by calculating the total number of searches required based on the given information about rows, columns, and search density. If the number of searches is greater than the sequential cutoff, I halve the number of searches and allocate two threads to handle this divided workload. Each thread checks if its assigned search count is still greater than the cutoff.

   This iterative division process continues until the number of searches for each thread becomes lower than the sequential cutoff. When the number of searches falls below the sequential cutoff, I create individual instances of the search class, with a random row and column to start at. The find valley method is then called, which finds the smallest value from our start point. If any search instance uncovers a lower value than what's currently stored in a shared variable, that new lower value replaces the previous one.

   The result that each thread finds is then compared to each other, and the lower value is then returned as the global minimum.

4. Experiment Design:

We will be comparing the performance in terms of time of the serial algorithm and the parallel algorithm, to do this we have created a standard set of inputs which will be given to both the parallel algorithm and the serial algorithm, to ensure fairness. The range of x and y values will be fixed at 0 to 1000. Each of these algorithms will be searching for valleys in a terrain that is defined by the Rosebrock function, this is to ensure accuracy, as the function only has one global minimum.

The dataset we will be using is:

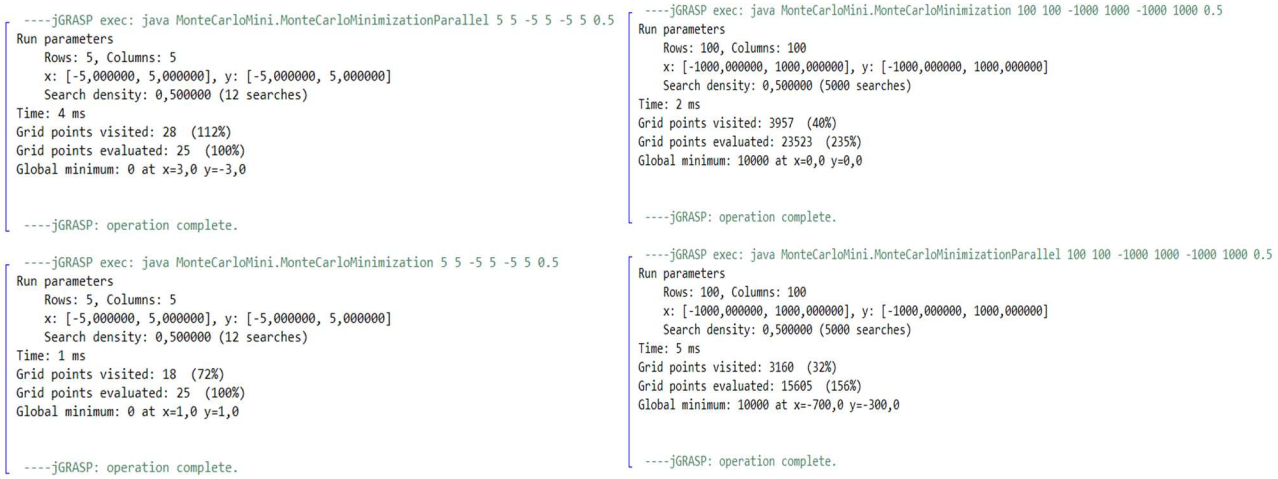
1) 10 rows;10 columns and 0.1 search density
2) 50 rows;50 columns and 0.2 search density
3) 100 rows;100 columns and 0.3 search density
4) 200 rows; 200 columns and 0.4 search density
5) 500 rows; 500 columns and 0.5 search density
6) 1000 rows; 1000 columns and 0.6 search density
7) 2000 rows; 2000 columns and 0.7 search density
8) 5000 rows; 5000 columns and 0.8 search density
9) 7500 rows; 7500 columns and 0.9 search density
10) 10000 rows; 10000 columns and 1 search density

We expect that as the workload gets larger the speed up should approach the number of cores in the machine we are testing on.

5. Data Collection:

I have developed a Python script to facilitate data collection by evaluating our standard inputs using both the parallel and linear java files. The process involves supplying a standard input to one of the algorithms and running it five times. During each run, the execution time of the algorithm is recorded. These time measurements are then gathered, and the median value is calculated to yield the final result. This approach ensures a comprehensive and statistically reliable assessment of algorithm performance.

We use a program to help us compare the results of the parallel and serial programs. This helps us check if our parallel approach is both correct and faster. In other words, we're making sure that the parallel approach gives us the right answers, just like the serial approach, and we're also checking if it's actually quicker. This way, we can be confident that our changes **Validated.**



Screenshot of the linear and parallel finding the same global minimum (ran on jGRASP).
All the code is in this git repository.

Results

Benchmarking:

We need a point of reference, to compare the performance of our parallel algorithm to. From running the linear algorithm 5 times each for a given standard input and taking the median, whilst the searches are being conducted on a terrain that is defined by the Rosebrook function.

The data collected from the linear algorithm:

This benchmark was ran on a 4 core intel i510300H

Number of rows	Number of columns	Search density	Time taken
10	10	0,1	1
50	50	0,2	1
100	100	0,3	1
200	200	0,4	3
500	500	0,5	12
1000	1000	0,6	57
2000	2000	0,7	335
5000	5000	0,8	2538
7500	7500	0,9	6513
10000	10000	1	13857

This benchmark was run on the UCT nightmare servers.

Number of rows	Number of columns	Search density	Time taken
10	10	0,1	4
50	50	0,2	5
100	100	0,3	6
200	200	0,4	11
500	500	0,5	34
1000	1000	0,6	119
2000	2000	0,7	624
5000	5000	0,8	5172
7500	7500	0,9	13588
10000	10000	1	32749

Performance Evaluation:

Parallel algorithm, data collected on an i51300H (4 cores)

Input id	Number of rows	Number of columns	Search density	Time for serial (ms)	Time for parallel(ms)	Speed up
1	10	10	0,1	1	3	0,3333333
2	50	50	0,2	1	3	0,3333333
3	100	100	0,3	1	5	0,2
4	200	200	0,4	3	7	0,4285714
5	500	500	0,5	12	18	0,6666667
6	1000	1000	0,6	57	77	0,7402597
7	2000	2000	0,7	335	219	1,5296804
8	5000	5000	0,8	2538	1227	2,0684597
9	7500	7500	0,9	6513	2473	2,6336433
10	10000	10000	1	13857	4736	2,9258868

Speed up graph.

The input ID is a combination of rows, columns, and search densities, gradually increasing in size.

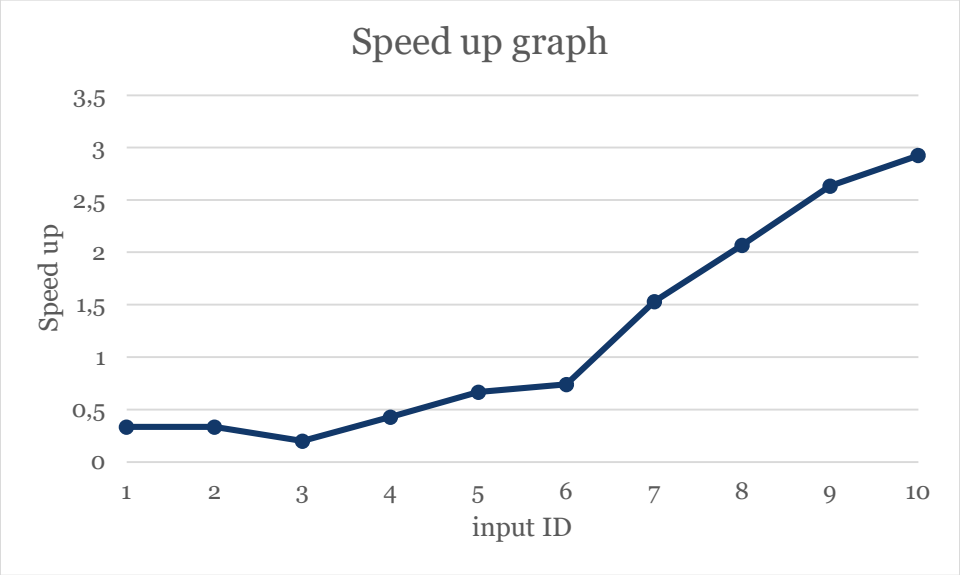


Figure 1: speed up graph for i51300H.

*As the workload increases the speed up approaches the number of core. But after this point we run into a heap size error, which prevents us from making the workload any larger

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at MonteCarloMini.TerrainArea.<init>(TerrainArea.java:25)
        at MonteCarloMini.MonteCarloMinimizationParallel.main(MonteCarloMinimizationParallel.java:90)
```

Parallel algorithm, data collected on nightmare servers.

Input id	Number of rows	Number of columns	Search density	Time for serial (ms)	Time for parallel(ms)	Speed up
1	10	10	0,1	4	11	0,3636364
2	50	50	0,2	5	12	0,4166667
3	100	100	0,3	6	14	0,4285714
4	200	200	0,4	11	16	0,6875
5	500	500	0,5	34	35	0,9714286
6	1000	1000	0,6	119	146	0,8150685
7	2000	2000	0,7	624	394	1,5837563
8	5000	5000	0,8	5172	2193	2,3584131
9	7500	7500	0,9	13588	4397	3,0902888
10	10000	10000	1	32749	8091	4,0475837

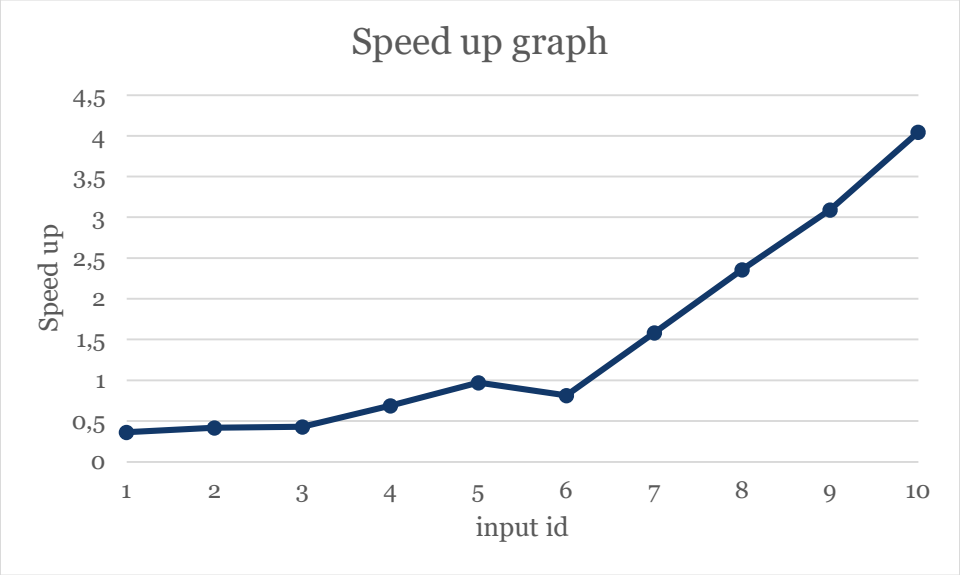


Figure 2: UCT nightmare speed up graph

*As the workload increases the speed up approaches the number of core. But after this point we run into a heap size error, which prevents us from making the workload any larger

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at MonteCarloMini.TerrainArea.<init>(TerrainArea.java:25)
        at MonteCarloMini.MonteCarloMinimizationParallel.main(MonteCarloMinimizationParallel.java:90)
```

Comparison and Analysis:

On both machines, it can be concluded that when the workload is low, the serial program runs faster, this is because the parallel program has a set overhead that makes it a lot slower than the serial in smaller workload cases.

in the case where the workload was high, we see that the parallel program runs significantly faster than the serial program, to a point were the speed up approaches the number of core until we run into the heap size ceiling.

In smaller workloads its best to use the serial approach, but in larger workloads its best to use the parallel approach. If not for the heap limitation, we would see that in both machines the speed up would approach the number of cores we have. This is because the workload is split among those cores.

Discussion

For this report we had an upbound limitation on how large we can make the grid size and the search density. This limitation was the heap size, past a certain point, making the grid or the search density larger would trigger an exception:

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at MonteCarloMini.TerrainArea.<init>(TerrainArea.java:25)
        at MonteCarloMini.MonteCarloMinimizationParallel.main(MonteCarloMinimizationParallel.java:90)
```

It is because of this exception that we chose the data size that we chose, with a maximum of 10000 rows and 10000 columns at a search density of one.

We went into this experiment expecting that when the work load increases, so will the speed up, and that the speed up will eventually approach the number of cores for a sufficiently large enough workload. This is exactly what was observed, for our four core machine at our highest workload we have a speed up of 3. For our 8 core machine at our highest workload we have a speed up if 4. If not for the heap limitation, we would have observer a gradual increase in the speed up, when we increase our workload to about the same value as the number of cores.

We observed that the speed up graph is less than 1 for small workloads as expected, and gradually, almost linearly increases as we increase the workload, as expected.

There is a race condition, as we will be accessing the same terrain object from multiple threads, this has had the effect showing not all grid points as having had been evaluated in the parallel program. It also has the effect of not giving the correct co-ordinates for which the global minimum was found.

Conclusion:

This report aimed to compare the time it takes to complete a task using serial means and using parallel means. In this report we used the monte Carlo algorithm to find the lowest point on a terrain that is defined by the Rosebrock function.

We can safely conclude that when running smaller workloads, i.e., when trying to find the global minimum in a smaller grid, with a smaller search density, the serial algorithm will run faster. When running much larger workloads, the parallel algorithm will run faster, almost as fast as many cores there are.

If the workload is large enough, it is absolutely worth the time and effort to parallelize the serial program, as one can expect a speed up close to the number of cores for a sufficiently large enough workload.