University of Cape Town

# CSC2002S Assignment PCP1 2023

Dylan Kuming: KMNDYL001

August 13, 2023

# Contents

# List of Figures

# Introduction and Methods

## 1.1 Parallelization Approach

In this assignment, the Java Fork-Join framework is used to speed up the algorithm. The parallelization tactic uses the Monte Carlo algorithm for finding the minimum (the lowest point) of a two-dimensional mathematical function $f(x, y)$ within a specified range.
Given the search density and the grid size, an array of search tasks is initialized. Each search task is designed to find valleys (local minimums) on the terrain. The tasks are divided using the 'ForkJoinPool'. The division occurs recursively: if the number of tasks in a section exceeds the sequential cut-off, it is split into two parts, and each part is processed in parallel. If the number of tasks is below the cut-off, the section is processed sequentially.
An implementation of an optimisation is that if a search reaches a point that a prior search has already visited, it stops as it would trace the same route to the identical local minimum.

## 1.2 Validation

To validate the parallel Monte Carlo minimization algorithm, first the sequential version will be run on several test terrains, and the results will be stored for comparison. Then, the parallel version will be run on the same terrains, and if the outputs match, this ensures that the parallel program is correct. The Rosenbrock function will be used for this process as it has a known minimum of height 0 at points $(1, 1)$. Multiple runs will be executed to identify and remove outliers for more accurate results.

## 1.3 Benchmarking

Benchmarking will be performed using a timer that records the time between the start and end of the search tasks. The start time is recorded right before initiating the parallel search and the end time is recorded right after completing the search. The performance of the algorithm will be assessed using various grid sizes and search densities in order to ensure sufficient coverage.
Tests will be run on two different machine architectures, and the benchmark will be repeated multiple times to ensure consistency.

## 1.4 Machine Architectures

The algorithms will be tested on two different machine architectures:

1. Machine A: This architecture features a 2.20 GHz Dual-Core Intel Core i7 processor.

2. Machine B: This architecture features a 1.40 GHz Quad-Core Intel Core i5 processor.

## 1.5 Problems/Difficulties Encountered

Deciding the appropriate sequential cut-off for dividing tasks is very important. If the sequential cut-off is set too low, then there will be excessive task division leading to high overhead from thread management. While setting the sequential cut-off too high will result in under-utilization of available cores as tasks are processed sequentially even when parallelism would be beneficial. Through trial and error, a sequential cutoff of 10000 was chosen.
Additionally, since the parallel solutions share a TerrainArea object, a potential race condition arises when threads write to the same grid spot. This will however be ignored since the race condition is minor, and protecting against it might result in the program being slower, without much benefit gained.

# Results

## 2.1 Figures

Two graphs are shown below in which figure 1 represents the speedup of the parallel program on a 2.20 GHz Dual-Core Intel Core i7 machine, while figure 2 represents the speedup of the parallel program on a 1.40 GHz Quad-Core Intel Core i5 machine.

The speedup versus grid size is plotted for different search densities of 0.1, 0.25, and 0.5. The terrain is kept at a constant size with parameters of $x_{\text{min}} = -10$; $x_{\text{max}} = 10$; $y_{\text{min}} = -10$; $y_{\text{max}} = 10$ and a `SEQUENTIAL_CUTOFF` $= 1000$. The grid size ranges from $500 \times 500$ to $10000 \times 10000$.

The speedup is calculated using the formula:

$$\text{Speedup} = \frac{T_{\text{Serial}}}{T_{\text{Parallel}}}$$

and is plotted using a logarithmic trend-line to ignore outliers which affect the results. Additionally, the tests for $T_{\text{Serial}}$ and $T_{\text{Parallel}}$ were run 6 different times, and the quickest time was recorded and used to calculate the speedup.

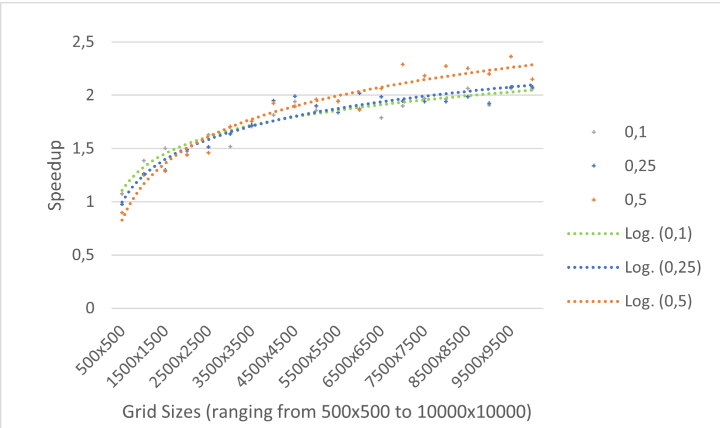


**Figure 1:** *Speedup versus Grid Size on 2.20 GHz Dual-Core Intel Core i7 machine with Varying Search Densities.*

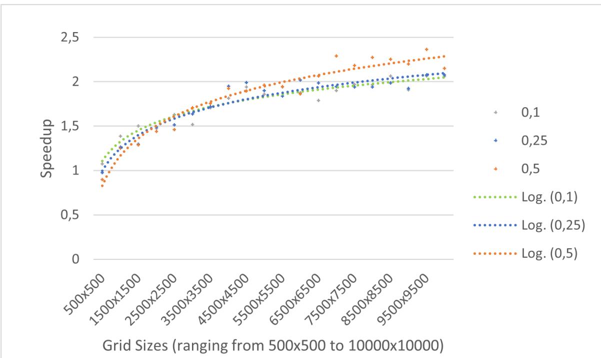


**Figure 2:** *Speedup versus Grid Size on 1.40 GHz Quad-Core Intel Core i5 machine with Varying Search Densities.*

## 2.2 Discussion

### 2.2.1 Range of grid sizes for optimal performance

The parallel program performs exceptionally well for medium to large grid sizes. For small grid sizes, the overhead of parallelization seems to offset the benefits. This is evident in the speedup graphs, where the speedup for small grid sizes is closer to 1, indicating near-sequential performance. As the grid size increases, the program starts to show significant speedups, reaching its peak at medium grid sizes. Beyond a certain point, however, there seems to be a saturation in speedup due to the inherent limits of parallelization and possibly memory bottlenecks.

For the 2.20 GHz Dual-Core Intel Core i7, the parallel program performs notably well for grid sizes from 4000x4000 and above, reaching peak performance around 7000x7000, and thereafter remaining somewhat constant until 10000x10000.

For the 1.4 GHz Quad-Core Intel Core i5, optimal performance is seen for grid sizes from around 6500x6500 and remaining relatively constant until 10000x10000.

### 2.2.2 Maximum speedup and comparison to the ideal

On the 2.20 GHz Dual-Core Intel Core i7, the maximum speedup achieved was approximately 2.362 when the search density was 0.5. This is greater than the ideal expected speedup of 2, considering it's a dual-core processor. This suggests super-linear speedup, possibly due the fact that while distributing tasks between the two cores, the data might be better located in cache memory, ensuring faster access and reducing wait times.

On the 1.4 GHz Quad-Core Intel Core i5, the highest speedup observed was approximately 1.768, which is a bit below the ideal expected speedup of 4 for a quad-core machine. The sub-optimal performance could be because the algorithm's parallel version might not be fully optimized to leverage the quad-core system's potential.

### 2.2.3 Reliability of Measurements

The measurements are relatively reliable. Each experiment was repeated 6 times to minimize the effects of outliers and the speedup is plotted using a logarithmic trend-line to further ignore outliers which effect the results. Furthermore, external factors like other running processes were controlled to the best extent possible to ensure consistent results.

### 2.2.4 Anomalies

There are instances, especially on the 2.20 GHz Dual-Core Intel Core i7 with search density 0.25, where the speedup drops slightly as the grid size increases, such as between the 4000x4000 and 5000x5000 grid sizes. This is perhaps due to the overheads associated with parallelization, which might occasionally offset the advantages of parallel execution.

On the 1.4 GHz Quad-Core Intel Core i5, for search density 0.25, there's a dip in speedup for the 5500x5500 grid size. This anomaly might be due to system-specific interruptions or inefficiencies in workload distribution among the four cores.

# Conclusions

Using parallelization in Java for this problem demonstrated significant potential:

1. **Performance Gains**: Both the 2.20 GHz Dual-Core Intel Core i7 and the 1.4 GHz Quad-Core Intel Core i5 showed improved speeds over the serial version with the 2.20 GHz Dual-Core Intel Core i7 machine even surpassing the expected dual-core speedup.
2. **Efficiency and Overheads**: While the parallel approach improved the performance of the algorithm, it also introduced complexities, such as race conditions. These overheads were ignored for our problem but could be more problematic in other algorithms and should always be accounted for.
3. **Development Complexity**: Parallelization increases code complexity, making development and debugging more challenging. This added complexity is however justified by the performance improvements.

In summary, for this problem, parallelization in Java is beneficial and offers performance advantages. However, the performance gains must be weighed up against the added complexities when considering this approach.

# Appendix

| Grid size | sequential cutoff = 0.1 | | | sequential cutoff = 0.25 | | | sequential cutoff = 0.5 | | |
|---|---|---|---|---|---|---|---|---|---|
| | serial (ms) | parallel (ms) | speedup | serial (ms) | parallel (ms) | speedup | serial(ms) | parallel(ms) | speedup |
| 500x500 | 44 | 41 | 1,0731707 | 43 | 44 | 0,977273 | 52 | 58 | 0,896552 |
| 1000x1000 | 133 | 96 | 1,3854167 | 178 | 141 | 1,262411 | 169 | 135 | 1,251852 |
| 1500x1500 | 360 | 240 | 1,5 | 275 | 212 | 1,29717 | 303 | 235 | 1,289362 |
| 2000x2000 | 484 | 329 | 1,4711246 | 516 | 347 | 1,487032 | 557 | 387 | 1,439276 |
| 2500x2500 | 818 | 503 | 1,6262425 | 794 | 525 | 1,512381 | 840 | 575 | 1,46087 |
| 3000x3000 | 1107 | 729 | 1,5185185 | 1184 | 723 | 1,637621 | 1334 | 782 | 1,705882 |
| 3500x3500 | 1615 | 946 | 1,7071882 | 1716 | 998 | 1,719439 | 1834 | 1045 | 1,755024 |
| 4000x4000 | 2206 | 1218 | 1,8111658 | 2461 | 1264 | 1,946994 | 2792 | 1452 | 1,922865 |
| 4500x4500 | 3091 | 1593 | 1,9403641 | 3217 | 1618 | 1,988257 | 3500 | 1847 | 1,894965 |
| 5000x5000 | 3765 | 2025 | 1,8592593 | 4002 | 2109 | 1,897582 | 4384 | 2236 | 1,960644 |
| 5500x5500 | 4676 | 2412 | 1,9386401 | 4986 | 2712 | 1,838496 | 5477 | 2819 | 1,942888 |
| 6000x6000 | 5617 | 2997 | 1,8742075 | 5886 | 2917 | 2,017827 | 6557 | 3524 | 1,86067 |
| 6500x6500 | 6551 | 3664 | 1,7879367 | 6946 | 3501 | 1,984005 | 8483 | 4109 | 2,064493 |
| 7000x7000 | 7796 | 4104 | 1,8996101 | 8149 | 4191 | 1,944405 | 10642 | 4645 | 2,291066 |
| 7500x7500 | 8986 | 4567 | 1,9675936 | 9427 | 4855 | 1,94171 | 12014 | 5505 | 2,18238 |
| 8000x8000 | 10428 | 5304 | 1,9660633 | 11006 | 5671 | 1,940751 | 14438 | 6355 | 2,271912 |
| 8500x8500 | 12368 | 5991 | 2,06443 | 12728 | 6416 | 1,983791 | 16242 | 7213 | 2,251768 |
| 9000x9000 | 13554 | 7107 | 1,9071338 | 14294 | 7433 | 1,923046 | 17871 | 8122 | 2,20032 |
| 9500x9500 | 15973 | 7748 | 2,0615643 | 17483 | 8415 | 2,0776 | 20893 | 8843 | 2,36266 |
| 10000x10000 | 17847 | 8662 | 2,0603787 | 18178 | 8750 | 2,077486 | 21243 | 9878 | 2,150537 |

**Figure 3:** *Table showing data from 2.20 GHz Dual-Core Intel Core i7 machine.*

| Grid size | sequential cutoff = 0.1 | | | sequential cutoff = 0.25 | | | sequential cutoff = 0.5 | | |
|---|---|---|---|---|---|---|---|---|---|
| | serial (ms) | parallel (ms) | speedup | serial (ms) | parallel (ms) | speedup | serial(ms) | parallel(ms) | speedup |
| 500x500 | 40 | 51 | 0,784314 | 37 | 42 | 0,880952 | 40 | 55 | 0,727273 |
| 1000x1000 | 124 | 143 | 0,867133 | 125 | 147 | 0,85034 | 136 | 138 | 0,985507 |
| 1500x1500 | 376 | 268 | 1,402985 | 373 | 315 | 1,184127 | 333 | 324 | 1,027778 |
| 2000x2000 | 563 | 389 | 1,447301 | 601 | 426 | 1,410798 | 601 | 440 | 1,365909 |
| 2500x2500 | 855 | 570 | 1,5 | 1001 | 621 | 1,611916 | 961 | 647 | 1,485317 |
| 3000x3000 | 1189 | 894 | 1,329978 | 1218 | 847 | 1,438017 | 1352 | 885 | 1,527684 |
| 3500x3500 | 1691 | 1200 | 1,409167 | 1775 | 1226 | 1,447798 | 1831 | 1222 | 1,498363 |
| 4000x4000 | 2131 | 1322 | 1,611952 | 2288 | 1588 | 1,440806 | 2575 | 1587 | 1,622558 |
| 4500x4500 | 2863 | 1937 | 1,478059 | 3042 | 1927 | 1,57862 | 3375 | 1960 | 1,721939 |
| 5000x5000 | 3411 | 2257 | 1,511298 | 3931 | 2325 | 1,690753 | 4445 | 2702 | 1,645078 |
| 5500x5500 | 4234 | 2791 | 1,517019 | 4453 | 2817 | 1,58076 | 4832 | 2932 | 1,648022 |
| 6000x6000 | 5239 | 3276 | 1,599206 | 5789 | 3490 | 1,658739 | 5715 | 3482 | 1,641298 |
| 6500x6500 | 6172 | 3989 | 1,547255 | 6877 | 3889 | 1,768321 | 7255 | 4294 | 1,689567 |
| 7000x7000 | 6862 | 4595 | 1,493362 | 7984 | 4681 | 1,705618 | 8074 | 5099 | 1,583448 |
| 7500x7500 | 8168 | 5625 | 1,452089 | 9130 | 5469 | 1,669409 | 9819 | 5662 | 1,734193 |
| 8000x8000 | 9589 | 6466 | 1,482988 | 10205 | 6467 | 1,578011 | 10723 | 6425 | 1,668949 |
| 8500x8500 | 11879 | 6780 | 1,752065 | 11439 | 7081 | 1,61545 | 12478 | 7130 | 1,75 |
| 9000x9000 | 12625 | 7590 | 1,663373 | 12540 | 7577 | 1,655009 | 13752 | 7640 | 1,8 |
| 9500x9500 | 13567 | 9029 | 1,502603 | 13775 | 8704 | 1,582606 | 15072 | 8420 | 1,79 |
| 10000x10000 | 15349 | 9285 | 1,653096 | 15814 | 9412 | 1,680195 | 16675 | 9213 | 1,81 |

**Figure 4:** *Table showing data from 1.40 GHz Quad-Core Intel Core i5 machine.*