Assignment PCP1: Parallelizing Monte Carlo Optimisation Function

Course Code:	CSC2002S
Author:	Bradley Carthew
Student No.:	CRTBRA002
Date:	4 April 2023

1. Methods

1.1. Parallelization Approach & Optimizations

In this section, we delve into the approach taken to parallelize the Monte Carlo minimization algorithm using Java's Fork/Join framework. A pivotal role is assumed by the SearchParallel class which extends *RecursiveTask<Integer[]*, enabling a SearchParallel object to become a task that can be divided into smaller subtasks. A SearchParallel object receives the number of searches, number of rows, number of columns, a TerrainArea object and a Rand object as parameters. Evidently, each SearchParallel task returns a result in the form of an integer array, constituting the local minimum's height and, x and y position in the grid.

Central to the parallelization approach is the *compute()* method of the SearchParallel class. This method asses the number of searches to be performed and makes a decision based on a predetermined sequential cutoff value. If the number of searches is less than or equal to this value, the *find_valleys()* method is invoked to find the local minimum of the current task, and depending on the remaining number of searches to be performed, the method calculates and compares the local minima of any additional SearchParallel objects in sequence, finally, returning the lowest local minimum. Alternatively, if the number of searches exceeds the sequential cutoff value, the program employs a divide and conquer approach to achieve parallelism. At this point, two SearchParallel objects are created, left and right, left is initiated using the *fork* operation, becoming a new task, while right is executed in the context of the current task using the *compute()* operation. The left object receives an integer result of the number of searches divided by two, and the right object receives the remaining number of searches. Once the left task returns a result, synchronized using the *join* operation, the height of the right task is compared to the left. The result constituting the lowest height is returned, and the program recurses back up the task hierarchy, finally returning the global minimum.

With regards to optimizations, the program utilizes only one SearchParallel object to compare and calculate the local minimum of searches sequentially, avoiding creating large amounts of SearchParallel arrays within the context of each task. A *reset_sequential_search()* method is used to reset the random starting position in the grid and the previous instances found local minimum, as well as assign the object a new Id. Additionally, an *AtomicInteger* is used to assign an Id to each search task, ie., a SearchParallel task that performs the *find_valleys()* operation, in a thread safe manner, and random starting positions are only calculated and assigned to these tasks. Finally, the sequential cutoff value was chosen to be the $\frac{number\ of\ searches}{number\ of\ cores \times 2}$, allowing the program to scale well on different machine architectures, and because it provided the best performance when compared to other sequential cutoffs.

1.2. Machine Architectures

In this section, we explore the machine architectures used to benchmark and evaluate the performance of the developed parallel program.

Specifications	Departmental (Nightmare)	Personal
Processor (CPU)	Intel(R) Xeon(R) CPU E5620 @ 2.40GHz, 8 cores.	AMD Ryzen 7 4800H with Radeon Graphics @ 2.90 GHz, 16 cores.
Memory (RAM)	8.0 GB (7.1 GB available)	16.0 GB (15.4 GB available)
Operating System	Ubuntu 22.04.2 LTS (22.04)	Windows 11 Home (22H2)
Java Runtime Environment (JRE)	Java (TM) SE Runtime Environment (build 17.0.6+9-LTS-190)	OpenJDK Runtime Environment Temurin-17.0.8+7 (build 17.0.8+7)

Figure 1: Hardware and software specifications of the two machines, departmental (nightmare) server and personal laptop, used to benchmark parallel programs performance.

Figure 1, above, demonstrates the use of two machine architectures with different core counts, and notably, different operating systems.

1.3. Validation

In this section, we discuss the approach taken to validate the parallel program. The Rosenbrock function, a commonly used optimization benchmark, was employed for validation purposes. This function possesses a global minimum of zero at coordinates (1, 1), a useful attribute considering the original trigonometric function used to calculate the height at each grid point could potentially have a global minimum at many different points on the grid. By analysing whether the program consistently identified this known global minimum, we could ascertain the correctness of the parallel program with respect to the serial program.

Evaluation involved comparing the output results of the parallel program to finite global minimum of the Rosenbrock function and comparing these results to those obtained by the serial program. A series of 10 test cases were executed, encompassing various grid sizes ranging from 10x10 to 5000x5000 units. The searches density was introduced randomly within the range of 0 to 1, with increments of 0.1. And finally, the tests were conducted 10 times, and the validation result expressed as the percent successful, i.e., a percentage of the number of correct results out of 10. The results from the validation tests can be seen in Appendix A, and demonstrate that the parallel program is accurate in its predictions. Notably, both the serial and parallel program do not always return a percentage success of 100%. This can be attributed to the random nature of the Monte Carlo minimization algorithm, especially when the searches density is low, and the grid size is large. In these scenarios, the probability of finding the correct global minimum is low because of the small number of searches being performed per grid point.

1.4. Benchmarking

In this section, we explore the methodology used for benchmarking the parallel program against the serial program. The serial program was profiled first (on the personal laptop) in order to benchmark the parallel program against it, and subsequently the parallel program was profiled on the two machines outlined in 1.2, above. Both the serial and parallel program were profiled using a series of test cases, ranging from grid sizes of 10x10 to 5000x5000 units, and searches density of 0.1 through to 0.9, with increments of 0.2 (see Appendix B for the test cases). Additionally, a secondary test set was utilized to assess the programs' performances on larger grid sizes and searches density of 0.5. Each test case was run five times, and an average execution time was recorded for each test case. Tests were performed in Git Bash, and Ubuntu terminal for the personal laptop and departmental server, respectively, and bash

scripts were utilized to automate the process. Finally, the parallel program was written and compiled using IntelliJ IDE.

1.5. Problems

In this section, we discuss any problems/difficulties that were found when benchmarking and validating the parallel program against its serial counterpart. Initially, when conducting profiling of the parallel and serial program on the personal laptop, tests had been performed using an Ubuntu-based environment utilizing WSL within the Windows operating system. However, after conducting tests on the departmental server, it was found that, the departmental server was outperforming both the serial and parallel program, particularly for smaller grid sizes and searches densities. Even after changing the configuration file for WSL to use the laptops full processing power and memory capacity, the tests still failed to perform as expected. As a result, the decision was made to transition away from WSL, and instead, testing was conducted using Git Bash, with OpenJDK JRE being installed within the Windows environment.

2. Results

In this section, the results of benchmarking the parallel program against the serial program are presented. The results are presented in the form of speedup graphs, each with two different plots, one for each machine tested on, i.e., the departmental server (nightmare) and a personal laptop, named according to core count. The results follow the experiments outlined in 1.4, above, and demonstrate how the parallel program scales with varying grid sizes and searches densities.

Additionally, the graphs include a threshold of “1” to help demonstrate where the parallel program showed improved execution time, and the x-axis represents the grid length, where the grid size is the grid length squared.

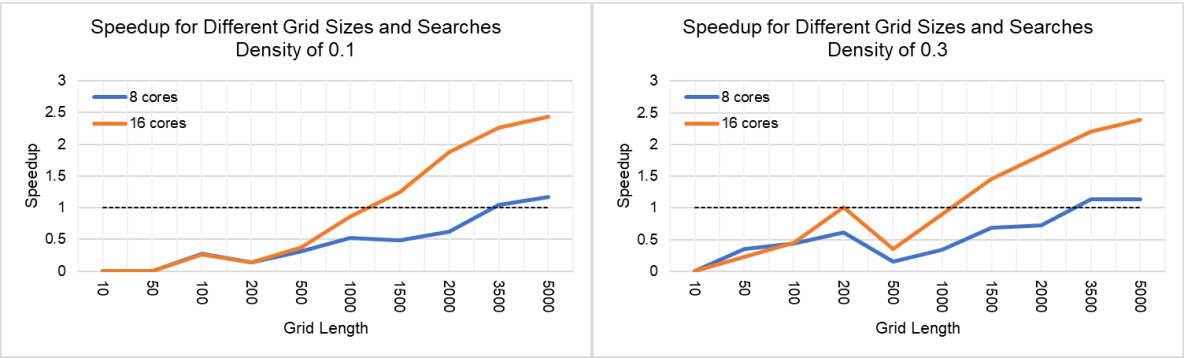

Figure 2: Speedup graphs for different grid sizes and searches density of 0.1 (left) and 0.3 (right).

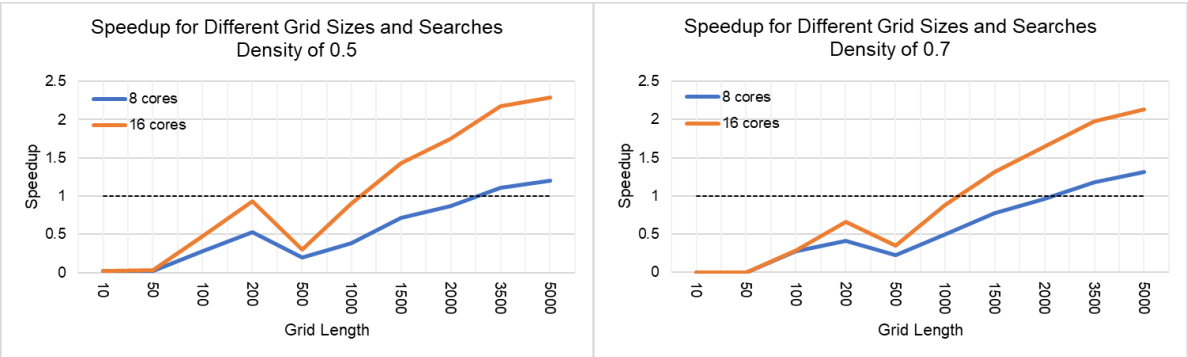

Figure 3: Speedup graphs for different grid sizes and searches density of 0.5 (left) and 0.7 (right).

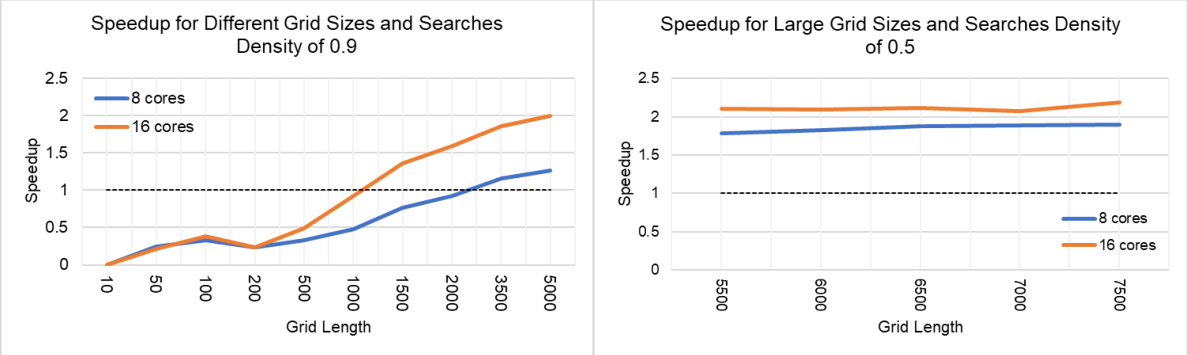

Figure 4: Speedup graphs for different grid sizes and searches density of 0.9 (left), and for large grid sizes with a searches density of 0.5 (right).

2.1. Discussion

In this section, we discuss the results obtained from benchmarking the parallel program against the serial program on two different machine architectures. Additionally, noting the reliability of the results obtained, and discussing any anomalies and why they occurred.

Looking at Figures 2 – 4, we can observe that the 16-core machine begins to achieve a speedup greater than one within a grid length range of 1000 – 1250 (1000x1000 – 1250x1250). On the other hand, the 8-core machine begins to achieve a speedup greater than one in a grid length range of 2000 – 3500 (2000x2000 – 3500x3500). Likewise, any results below these points are instead dominated by the serial program, which demonstrates the best performance. It's also worth noting, that although the number of searches increases with both grid size and searches density, the point at which a speedup greater than one is achieved doesn't seem to decrease, with respect to the grid length, in a linear manner. Instead, it appears to halt and oscillate at a grid length of around 1000 for the 16-core machine and around 2000 for the 8-core machine. Therefore, achieving no improvement below this point, indicating that no further speedup can be achieved for smaller grid sizes. Additionally, speedup increases from these aforementioned-points and begins to plateau and oscillate around a maximum speed up, as seen in the right graph in Figure 4, demonstrating speedup for large grid sizes and a searches density of 0.5. Consequently, the parallel program achieves the best speedup on the 16-core machine for grid sizes greater than 1250x1250 and a searches density greater than 0.1.

As mentioned, the speedup for the parallel program seems to plateau and oscillate around a maximum speed up for both machines. This maximum speedup being, around 2.2 for the 16-core machine, and 1.9 for the 8-core machine. Ideally, the maximum speedup expected for the parallel program should be equal to the number of cores on the machine, i.e., 8x for the departmental server, and 16x for the personal laptop. Evidently, the achieved speedups obtained during testing do not match the ideal speedups, as specified above. Instead, the speedup achieved for the 16-core machine is 7.2x smaller than that of the ideal, and for the 8-core machine the speedup achieved is 4.2x smaller than that of the ideal.

While these results offer a sturdy basis for drawing conclusions and initiating discussions, it's crucial to address certain factors that could potentially undermine the reliability of the measurements acquired during testing. First and foremost, when contrasting the outcomes of parallel programs on two distinct machine architectures, it's essential to consider the variations arising from differences in operating systems and hardware architectures. Although both machines run the same version of the parallel program, the way that the operating system manages and schedules the threads created by the JVM will be different when comparing Windows and Ubuntu Linux systems. Therefore, it cannot be said that the performance of multithreading on two different operating systems can be likened. Additionally, the programs are running on two machines with different hardware architectures, which means different

CPU clock speeds, instruction pipelining, memory bandwidth and cache hierarchy, to name a few, could all potentially have an affect on the performance of the program. Therefore, comparing two machines solely based on the number of cores available might not provide thorough insight into their performances. Next, we scrutinize the performance of individual threads and how tests might differ from one another. When using threads, it cannot be guaranteed that the threads created by the JVM will run without contending with other threads from different programs being run on the machine at the same time. Although making sure that other programs are closed on the machine could help, this does not ensure non-contention and threads could be subject to unpredictable and inconsistent delays. For example, page faults, cache misses and the scheduled quantum finishing, could all result in threads being delayed and anomalies occurring inconsistently across testing scenarios. Finally, these tests are not exhaustive, and it cannot be said for certain that the same discussions/conclusions could be made for grid size of different shapes, i.e., rectangular grid shapes, and for changing x and y limits. It also should be noted that the use of average to find the execution time per test case can be skewed by outliers and produce inaccurate measurements due to anomalies occurring.

Finally, looking at Figures 2 – 4, there seems to be a trend where the speedup for both machines increase linearly, then begins to decrease linearly and finally increases exponentially, surpassing the performance of the serial program. For example, in the left plot of Figure 3, the speedup increases within a grid length range of 50 – 200 (50x50 – 200x200), then decreases between 200 – 500 (200x200 – 500x500), and thereafter increases exponentially to the maximum speed up. This is attributed to the overhead required to setup of the parallel program. Initially, when dealing with smaller problem sizes, the overhead needed to set up the parallel program is quite modest, hence the speedup increase. However, the overhead is still large enough and the problem size small enough to not see an improved performance over the serial program. Thereafter, as the problem size increases, the overhead required to setup the parallel program becomes too large, and the speedup decreases, demonstrating the serial programs dominance for smaller problem sizes. Once, the problem size becomes too large for the serial program, the execution time increases, and the overhead required to setup the parallel program is no longer inhibiting an increase in speedup, hence the parallel program scales exponentially with problem size until the maximum speedup is reached.

3. Conclusions

In conclusion, the results obtained from benchmarking the parallel program against its serial counterpart across two distinct machine architectures, has produced valuable insights into the behaviour and performance of parallelization. Figures 2 - 4 offered a visual narrative of how the parallel program's speedup evolved with varying gride sizes and searches densities.

The following observations and considerations can be drawn from the discussion had in 2.1, above.

3.1. Speedup Trends and Performance Domains

The most notable observation is the consistent trend in speedup on both the 16-core and 8-core machines. The trend entails an initial linear speedup increase, followed by a linear decrease, and eventually culminating in an exponential increase that surpasses the performance of the serial program. A point of interest is the juncture at which the speedup surpasses one. On the 16-core machine, this occurs within a grid length range of 1000 – 1250, while the 8-core machine achieves this between 2000 – 3500. Results below these points exhibit the serial program’s dominance over smaller problem sizes, while results above these points exhibit the parallel program’s ability to scale reasonably well with larger problem sizes.

3.2. Non-Linear Progression

Remarkably, the progression from a speedup less than one to greater than one doesn't exhibit a straightforward linear correlation with increasing grid size. Instead, it seems to stabilize and oscillate around specific grid lengths. This non-linear behaviour suggests that achieving speedup improvements requires reaching a point where the parallel programs benefits outweigh the initial overhead costs.

3.3. Maximum Speedup and Anomalies

Although one might expect the maximum speedup to align with the number of cores, the attained speedup fell short of the ideal. The maximum speedup of 2.2 for the 16-core machine and 1.9 for the 8-core machine highlights a gap between the expected and achieved performance. Possible factors contributing to this discrepancy include the intricacies of operating systems, hardware architectures and thread contention, which highlight the complexity of achieving perfect parallel scaling. Additionally, with respect to Amdahl’s law, the maximum speedup is influenced by the portion of the code has not been parallelized, the overhead required to setup the parallel program and any optimizations not considered.

3.4. Reliability and Considerations

Discrepancies arising from differences in operating systems and hardware architectures can introduce variations between machine performances. Furthermore, the potential for unpredictable thread interactions, including delays from external factors, challenges the consistency of testing scenarios. Additionally, the choice of metrics, such as averaging execution times, can lead to inaccuracies due to anomalies.

In summary, the investigation into the parallel program's performance demonstrated the relationship between problem size, parallelization overhead, and system intricacies. The non-linear patterns observed indicate the necessity of larger problem sizes to unlock the parallel program's potential. While, these findings have been insightful, its clear that parallelization is multifaceted, with further investigations warranted for various problem shapes and limits. Finally, when the parallel program handles substantial problem sizes, the benefits from multiple cores, and potential speedup outweighs the complexities from overhead, and parallelization becomes necessary to enhance the performance of the Monte Carlo minimization function.

A. Results from Validation of Parallel Program using Rosenbrock Function

Table 1: Percentage success of the parallel and serial program when evaluating the global minimum of the Rosenbrock Function.

	Test Cases	Parallel Success (%)	Serial Success (%)
1	10 10 0.0 10.0 0.0 10.0 0.1	100	100
2	50 50 0.0 50.0 0.0 50.0 0.4	100	100
3	100 100 0.0 100.0 0.0 100.0 0.3	100	100
4	200 200 0.0 200.0 0.0 200.0 0.5	90	100
5	500 500 0.0 500.0 0.0 500.0 0.2	100	100
6	1000 1000 0.0 1000.0 0.0 1000.0 0.8	100	100
7	1500 1500 0.0 1500.0 0.0 1500.0 0.9	100	100
8	2000 2000 0.0 2000.0 0.0 2000.0 0.7	100	100
9	3500 3500 0.0 3500.0 0.0 3500.0 0.6	100	100
10	5000 5000 0.0 5000.0 0.0 5000.0 0.1	100	80

B. Test Cases for Benchmarking the Parallel Algorithm

In Table 2, below, the underlined text, SD, represents the searches density which varied from 0.1 to 0.9 in increments of 0.2.

Table 2: Test cases used to benchmark the parallel program on different machines and against the serial program.

Different Grid Sizes and Searches Densities	Large Grid Sizes and Searches Density of 0.5
10 10 0.0 10.0 0.0 10.0 SD	5500 5500 0.0 5500.0 0.0 5500.0 0.5
50 50 0.0 50.0 0.0 50.0 SD	6000 6000 0.0 6000.0 0.0 6000.0 0.5
100 100 0.0 100.0 0.0 100.0 SD	6500 6500 0.0 6500.0 0.0 6500.0 0.5
200 200 0.0 200.0 0.0 200.0 SD	7000 7000 0.0 7000.0 0.0 7000.0 0.5
500 500 0.0 500.0 0.0 500.0 SD	7500 7500 0.0 7500.0 0.0 7500.0 0.5
1000 1000 0.0 1000.0 0.0 1000.0 SD	
1500 1500 0.0 1500.0 0.0 1500.0 SD	
2000 2000 0.0 2000.0 0.0 2000.0 SD	
3500 3500 0.0 3500.0 0.0 3500.0 SD	
5000 5000 0.0 5000.0 0.0 5000.0 SD	