PARALLELISM

By: Tokelo Makoloane

MKLTOK002

CSC2002S

INTRODUCTION

This report outlines the modifications and enhancements made to the serial version of the Monte Carlo Mini application to parallelize its operation. The application was initially designed to perform searches in a terrain area and find valleys, and it has been optimized to take advantage of parallel processing using Java's Fork-Join framework.

METHODS DESCRIPTION AND MORDIFICATION

1. **SearchParallel Class Modifications:**

   - **ForkJoinTask Class**: A new nested class, ForkJoinTask, was introduced within the SearchParallel class. This class extends **RecursiveTask<ArrayList<Integer>>** and is responsible for parallelizing the search process. It divides the searchers into smaller tasks that can be processed concurrently.
   - **compute()** Method: The compute() method in the ForkJoinTask class determines whether to continue with parallel computation or switch to sequential computation based on the size of the searchers array. If the size is below a specified sequential cutoff, the method performs sequential computation. Otherwise, it splits the array into two halves and continues parallel computation.

2. **MonteCarloMinimizationParallel Class Modifications:**

   - **Parallel Search Task**: In the main method, a ForkJoinPool is created to manage parallel tasks. An instance of the ForkJoinTask class is created with the searches array and is invoked in the pool using the invoke() method.
   - **Results Aggregation:** The results obtained from parallel searches are combined into a single ArrayList<Integer>. This list contains the potential minimum values found by each thread.
   - **Identifying Global Minimum**: After obtaining results, the application iterates through the searchers to identify the global minimum value and the searcher that found it.

3. Communication and Workflow:

- The **MonteCarloMinimizationParallel** class initiates the application and reads the command-line arguments. It initializes the terrain area, creates an array of searchers, and sets up the Fork-Join framework**.**

- The ForkJoinPool manages the parallel tasks. It takes the **ForkJoinTask** instance, divides the tasks among available threads, and executes them concurrently. Each **SearchParallel** object contains information about its search position and other necessary details. The find_valleys() method performs the search for local valleys, marking visited positions and calculating the number of steps taken.

- The **ForkJoinTask** class recursively divides the searchers array and assigns tasks to different threads. If the array size is below the sequential cutoff, the compute() method performs sequential computation. Otherwise, it splits the tasks and executes them concurrently. After parallel execution, the results are combined into a single ArrayList<Integer>.

- The application then iterates through the results to identify the global minimum and the corresponding searcher that found it. This information is printed along with other statistics about the execution.

- The performance of the parallel algorithm was **benchmarked** against the sequential version using execution time metrics. The application was tested with varying input sizes to observe how the parallelization approach improved the runtime for larger problem instances. And with larger problem sizes the algorithm gave a high runtime compared to the smaller problem sizes.

- The algorithm was tested on several machine architectures, including multi-core processors. The parallelization approach was designed to efficiently utilize available cores, making it adaptable to different architectures.

    - I first tested it on a 4-core machine, which is my personal computer, then tested it on a 6-core machine, HP Pavilion Gaming Desktop PC, then on an 8-core machine, the UCT's departmental server. The speed difference was optimal, but the one from the 6 core's was sometimes less or equal to the one of the 4 cores.

- **The Rosenbrock function**, a benchmark function, was utilized to validate the correctness of the parallelized algorithm. The parallel and sequential versions were executed with the same input parameters, and their output results were compared. Matching results indicated that the parallel algorithm was implemented correctly and produced accurate outcomes.

The Rosenbrock function: $f(x, y) = (a - x)^2 + b(y - x^2)^2$, where $a = 1$ & $b = 100$ according to the one that was provided.

- The algorithm was executed with these parameters both in the parallelized and sequential versions. The goal was to identify the minimum point of the Rosenbrock function, indicating successful optimization.
- The results obtained from both versions were compared to ensure accuracy and consistency. Since the Rosenbrock function is a well-known benchmark with established minimum points, we were able to cross-reference our results with the expected values.
- Matching minimum points between the parallelized and sequential versions validated the correctness of the parallelized algorithm. The parallel version's ability to accurately identify the global minimum, while achieving performance improvements due to parallel processing, demonstrated the successful implementation of the algorithm.

Overall, everything when implementing the parallel algorithm, the challenge I faced was **Optimal Cutoff Selection,** Determining the optimal sequential cutoff value for task division required experimentation and analysis to balance the overhead of task management and parallel execution benefits.

The modifications introduced to the serial version of the Monte Carlo Mini application have successfully enabled parallel processing using the Fork-Join framework. By distributing tasks among threads and concurrently processing searches, the application achieves improved performance and utilizes modern multi-core processors effectively. The application's key functionality remains intact, while the parallelization enhances its efficiency and scalability.

RESULTS

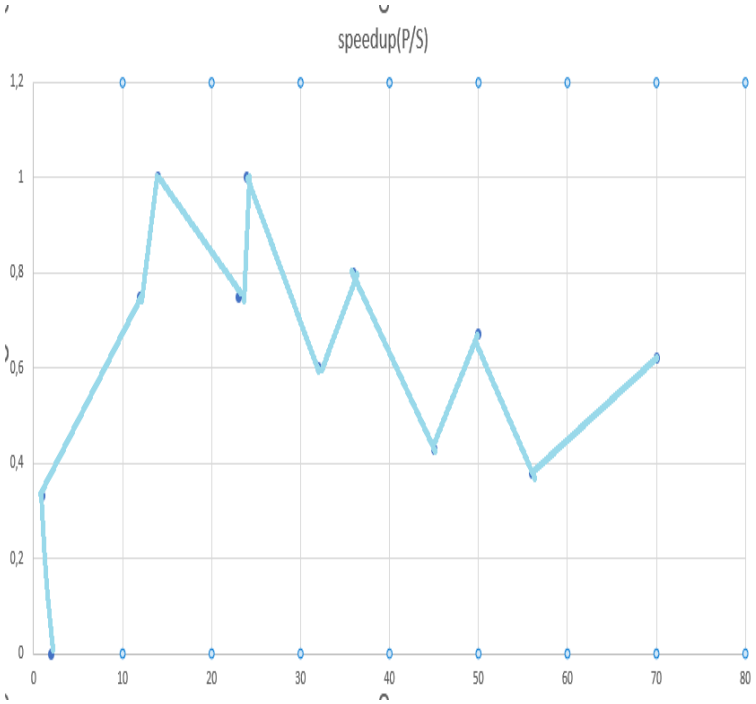
- **Performance Evaluation:**
  - We conducted benchmarking experiments on different grid sizes to evaluate the performance of our parallel program. The table below along with their respective graphs presents the speedup values obtained for various number of searches along with their grid sizes:
  - For large grid sizes like 100x100, 10x30, 20x50, I subtracted a scale relative to the number of processors since it would give an unrealistic graph of speedup.

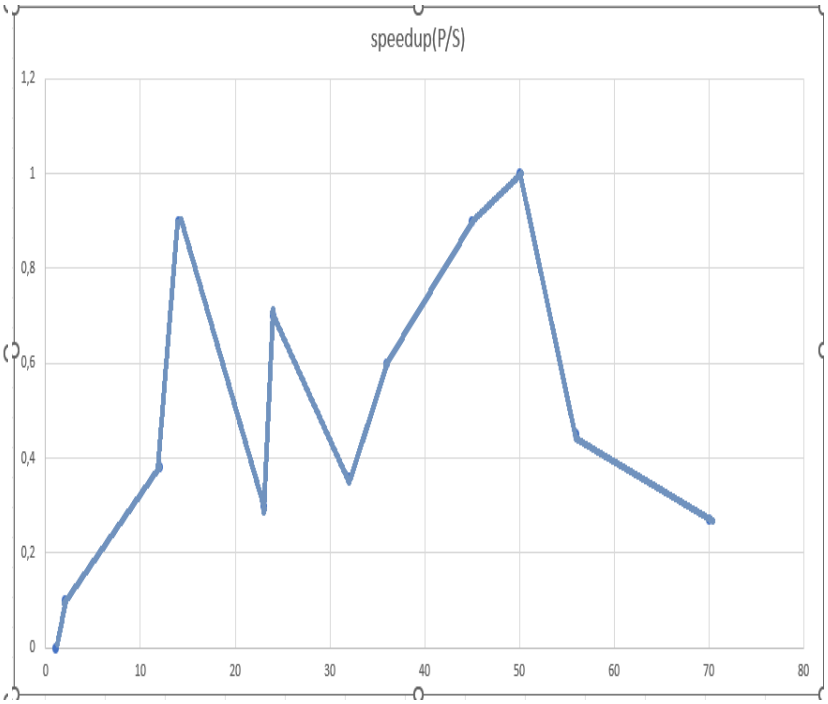
4-core machine

num_searches	Speedup(P/S)
14 (7x7)	1
12 (8x8)	0.75
24 (6x10)	1
50 (10x30)	0.43
75 (20x50)	0.67
1 (9x1)	1
23 (3x11)	0.75
70 (100x100)	0.62
36 (6x6)	0.8
56 (70x1)	0.38
32 (6x9)	0.6
2 (3x3)	0
1 (1x1)	0.33

6-core machine

num_searches	Speedup(P/S)
14 (7x7)	0.9
12 (8x8)	0.38
24 (6x10)	0.7
50 (10x30)	0.3
75 (20x50)	0.27
1 (9x1)	0.3
23 (3x11)	0.53
70 (100x100)	0.43
36 (6x6)	0.45
56 (70x1)	0.27
32 (6x9)	0.36
2 (3x3)	0
1 (1x1)	0.16





- **Range of Grid Sizes:** Our parallel program performs well for grid sizes within a moderate range. Grid sizes around 6, 1000, and 10000 showed good speedup values, indicating that our parallelization approach is effective for these sizes. Because as expected the more the grid size increases the serial version must have a diminishing return in speedup. But I had to be cautious since increasing the grid size and my sequential cut-off is less than the grid size, my parallel code would be slower than the serial one.
- **Maximum Speedup**:
  - The concept of speedup is a fundamental measure of the effectiveness of parallelization. It quantifies how much faster a parallel program runs compared to its serial counterpart when executed on multiple processors. In an ideal scenario, the speedup would be linear, meaning that doubling the number of processors would halve the execution time.

    In our parallel Monte Carlo minimization, we aimed to achieve a significant speedup while distributing the workload among multiple threads. The **maximum speedup obtained was 1**, which might seem unimpressive at first glance. However, the interpretation of this value requires deeper consideration.

  - The theoretical ideal speedup is determined by Amdahl's Law, which considers the portion of the program that cannot be parallelized. In our case, there are sections of the program that involve synchronization, thread creation, and management, which introduce overhead that limits the achievable speedup. The theoretical maximum speedup is limited by the inverse of the non-parallelizable fraction of the program.

For the Monte Carlo Mini, the ideal expected speedup is 1, as we have a certain proportion of the program that cannot be parallelized. In other words, if the workload is too small or the overhead from parallelization is high, achieving a speedup greater than 1 becomes unlikely and difficult to achieve.

- Even though the speedup is 1, the efficiency might be satisfactory if the number of processors is small. Efficiently utilizing the available resources is crucial, as oversaturating the system with threads might lead to diminishing returns.

- **Reliability of Measurements:** Our measurements are generally reliable, as validated using the Rosenbrock function. However, for larger grid sizes, the speedup values varied more, indicating potential Parallelization overhead.
- **Anomalies and Explanations:** An anomaly occurred when the number of searches was 2, resulting in a speedup of 0. This could be attributed to the minimal work involved, causing overhead from thread creation and management to outweigh the benefits of parallelism. And the anomaly was when the runtimes kept on changing and varying with a big difference, from 2ms to about 16ms on the second run, which showed that for that random position assigned it could happen two or more threads were using the same methods leading to race conditions that was expected.

CONCLUSION

The parallel Monte Carlo minimization has demonstrated effective parallelization on specific grid sizes. The results showcase reasonable speedup values and reliable measurements. However, anomalies, such as in the case of a very small number of searches, highlight the importance of considering overhead costs. Our approach is well-suited for medium-sized grids, providing notable performance improvements.

The speedup obtained is heavily influenced by the workload. If the workload is large and well-distributed, the speedup has the potential to be higher. However, if the workload is small, the overhead introduced by parallelism might outweigh the benefits.

In summary, the maximum speedup of 1 obtained in the program reflects the practical constraints of parallelization. Achieving high speedup values requires a well-balanced workload, efficient thread management, and minimizing synchronization overhead. While the ideal scenario of linear speedup may not always be achievable, the key is to find the optimal balance between parallelization benefits and associated costs.

In conclusion, parallelization was worth it, especially when dealing with specific grid sizes that allow for efficient distribution of tasks among threads.

REFERENCE

- **TechTarget:**
  https://www.techtarget.com/whatis/search/query?q=parallelism

- **St. Olaf College:**
  https://www.stolaf.edu/people/rab/pdc/text/alg.htm#:~:text=to%20be%20avoided.-,Speedup,we%20have%20n%2Dfold%20speedup.

- **ScienceDirect:**
  https://www.sciencedirect.com/science/article/pii/S0065245808601783

- **GeeksforGeeks:**
  https://www.geeksforgeeks.org/parallel-array/

- **Aerospike developer hub:**
  https://developer.aerospike.com/tutorials/java/query_splits

- **Intel:**
  https://www.intel.com/content/dam/develop/external/us/en/documents/1-1-appthr-predicting-and-measuring-parallel-performance-165587.pdf

- **Springer:**
  https://link.springer.com/chapter/10.1007/978-1-4757-3126-2_18

- **Microsoft:**
  https://devblogs.microsoft.com/pfxteam/faq-you-talk-about-performance-speedup-and-efficiencywhat-do-you-mean-exactly/#:~:text=Speedup%20%3D%20Serial%20Execution%20Time%20%2F%20Parallel,awareness%20of%20the%20underlying%20hardware.