Your Report Title Here

Your Name

October 21, 2023

1 Analysis of Original Application

1.1 Description

The original application selected for this project was the implementation of the Merge Sort algorithm in C#. Merge Sort is a renowned divide-and-conquer sorting algorithm that recursively divides an array into two halves, sorts each half, and merges them back in a sorted order. This algorithm was chosen because of its computational intensity, especially when dealing with substantial datasets. Its time complexity, $O(n \log n)$, can be relatively slow for large datasets. The divide-and-conquer nature of the Merge Sort algorithm, inherently, makes it a promising candidate for parallelization. Given the potential, the goal was to harness parallel processing capabilities to enhance the sorting speed and reduce execution time.

This specific sequential version of Merge Sort was initially implemented from scratch for this project and can be found on my GitHub. As a part of CAB301, I had been introduced to this algorithm, which laid a foundation for this project.

1.1.1 Two main phases constituted the application's analysis:

- Profiling the Sequential Implementation: A Stopwatch was integrated to measure the time taken in data generation and sorting, providing concrete metrics to establish the application's baseline performance. The initial version, though functional, had noticeable performance bottlenecks, especially when handling substantial data arrays.

- Identifying Potential for Parallelization: Recognizing the computational heft of Merge Sort, especially for expansive datasets, it was hypothesized that sorting various segments of the data array in parallel could drastically decrease the overall execution time. The inherent structure of Merge Sort—its division of data—offered an organic path towards parallelization.

1.2 Issues/Bottlenecks Identified

1. **Sequential Processing:** The original merge sort algorithm was executed in a sequential manner, which doesn't exploit the potential benefits of modern multi-core processors. Given the divide-and-conquer nature of merge sort, this was a significant area to introduce parallelism for better performance.

2. **Recursive Overhead:** The merge sort's recursive nature can lead to additional overhead in managing function calls, especially with larger datasets. Parallel

processing can help distribute this load across cores, reducing the time taken per recursion.

3. **Data Copying and Merging:** The operation of copying array segments and merging them in the merge sort increases computational overhead. Parallelizing these operations, especially the merge operation, can lead to substantial performance gains.

4. **Non-Adaptive Nature:** The original algorithm didn't adapt to the available system resources. An optimized parallel version can be adaptive by determining the number of cores available and then segmenting the data accordingly for concurrent processing.

5. **Merging Limitation in Optimized Version:** Despite the introduction of parallelism in the sorting segments in the optimized version, the merging process was still sequential, representing an area that could be further improved through parallel merging techniques.

1.3 Data and Control Dependencies

- **Data Dependency:** Discuss any data dependencies that were discovered.
- **Control Dependency:** Discuss any control dependencies that were found.

1.4 Before and After Profiling Results

Before: Insert detailed profiling results before optimization.
After: Insert detailed profiling results after optimization.

2 Use of Tools and Techniques

2.1 Tools Used

1. Tool 1: Describe how you used this tool.
2. Tool 2: Describe how you used this tool.

2.2 Techniques Implemented

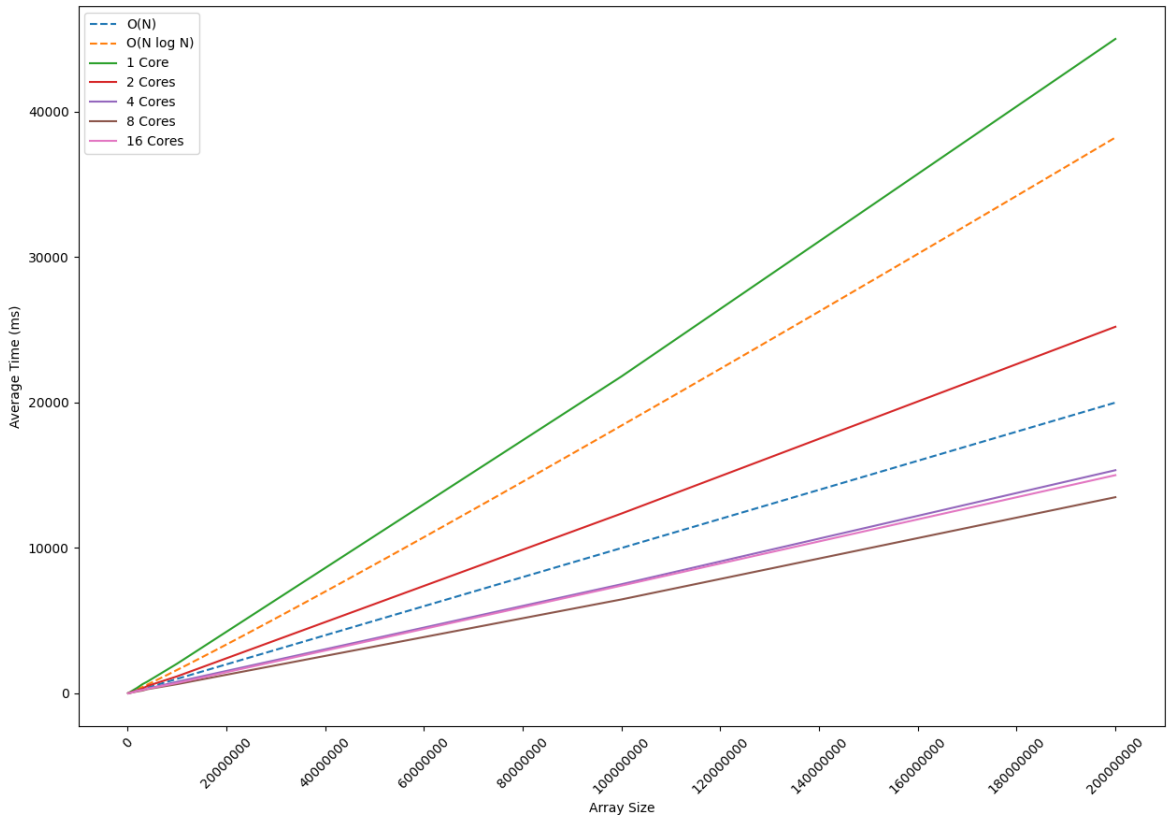
1. Technique 1: Discuss the first parallel programming technique you applied.
2. Technique 2: Discuss the second parallel programming technique you applied.

3 Optimal Speedup

3.1 Description

Provide a brief description of how the speedup was achieved.

3.2 Speed-up Graph



3.3 Performance Metrics

Before Optimization: Mention metrics such as execution time, throughput, etc.
After Optimization: Mention improved metrics.

4 Overcoming Barriers

4.1 Major Barriers Identified

1. Barrier 1: Describe the first major barrier you faced.
2. Barrier 2: Describe the second major barrier you faced.

4.2 Solutions Implemented

1. Solution to Barrier 1: Discuss how you overcame the first barrier.
2. Solution to Barrier 2: Discuss how you overcame the second barrier.

4.3 Before and After Code Snippets

```
% Insert code snippet before changes.
```

```
% Insert code snippet after changes.
```

5 Report

5.1 Introduction

Provide a brief introduction to your report.

5.2 Body

5.2.1 Section 1

Discuss the main content of your report.

5.2.2 Section 2

Expand on additional findings or insights.

5.3 Conclusion

Provide a summary of your findings and insights.

5.4 Recommendations

Provide any recommendations for future improvements or optimizations.