PDC PROJECT REPORT - Analyzing Sorting Algorithms in Serial, OPENMP, and MPI

21k-3211 Shayan Haider, 21k-3212 Ahmed Ali, 21k-4680 Taha Hassan
Section F

1 Introduction

The goal of this project is to conduct a comprehensive analysis of ten different sorting algorithms: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, Bitonic Sort, Comb Sort, Counting Sort, Heap Sort, and Radix Sort. The study involves implementing these algorithms in three different paradigms: serial, OpenMP (shared memory multiprocessing), and MPI (Message Passing Interface for distributed memory systems). The primary objective is to evaluate their performance concerning the number of threads/processes and the time taken to sort a particular dataset.

2 Problem Statement

Sorting algorithms play a pivotal role in numerous computing applications. However, their performance can vary significantly based on the size of the dataset and the computing environment. This project aims to compare and analyze the efficiency of ten sorting algorithms under different parallelization paradigms. Understanding how these algorithms perform in serial, shared memory, and distributed memory environments will provide insights into their scalability, speed, and suitability for varying computational architectures.

3 Algorithm Overview

3.1 Serial Implementation

– Bubble Sort: Compares adjacent elements and swaps them if they are in the wrong order. Repeats until the entire list is sorted.

– Selection Sort: Finds the minimum element from the unsorted part and places it at the beginning. Repeats until the entire list is sorted.

– Insertion Sort: Builds the final sorted array one item at a time by repeatedly taking the next element and inserting it into the correct position.

– Merge Sort: Divides the array into smaller parts, sorts them, and then merges them back together.

– Quick Sort: Chooses a ”pivot” element and partitions the array around the pivot, sorting smaller elements to the left and larger elements to the right.

– Bitonic Sort: A parallel sorting algorithm designed for sorting bitonic sequences, which involves comparing and swapping elements in a pre-defined order.

– Comb Sort: Improves on Bubble Sort by eliminating large values at the end of the list efficiently.

– Counting Sort: Assumes that the input consists of integers within a specific range and counts the number of occurrences of each value.

– Heap Sort: Builds a heap from the input data and repeatedly extracts the minimum element to obtain a sorted list.

– Radix Sort: Sorts elements by first grouping them based on individual digits and then combining them back together.

3.2 Parallel Implementations

- OpenMP: Utilizes shared memory parallelism by creating multiple threads to execute parts of the sorting algorithm simultaneously.
- MPI: Utilizes message passing for distributed memory systems, where processes communicate and collaborate to perform sorting operations on different segments of the dataset.

3.3 Time Analysis

To ensure manageable evaluation and avoid time-consuming core dump issues, we limited our comparison of serial, OpenMP, and MPI executions to a 200KB dataset. This decision allowed for a smoother assessment of performance across these execution paradigms. However, analysis on larger sets of data is also possible. We took 3 sorting algorithms out of 10 to analyze 3 cases where we can analyze performance in serial and parallel:

3.3.1 Bubble Sort

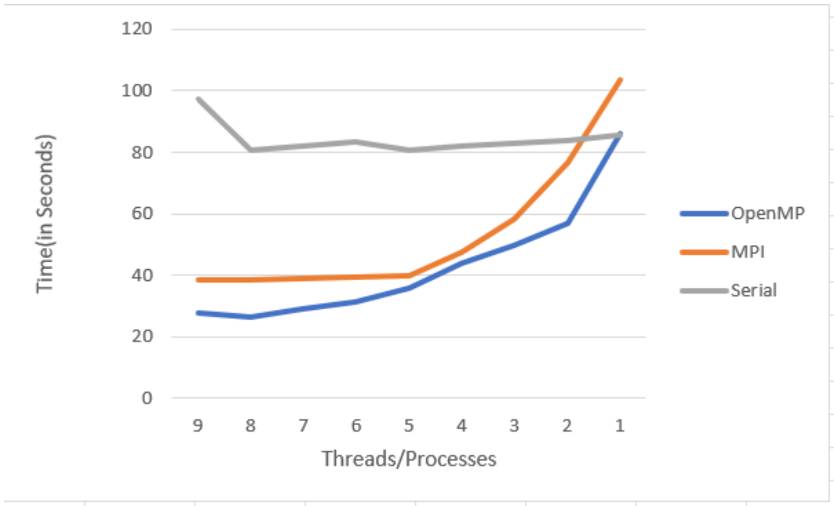

Figure 1: Bubble Sort in Serial and Parallel

- Serial vs OpenMP Execution Time Analysis: Observing Figure 1, it's evident that the average execution time for serial bubble sort stands at 84.31383478 seconds. In contrast, utilizing OpenMP, the average execution time notably decreases to 42.98561 seconds. The increase in execution time as the number of threads decreases can be attributed to the efficiency gained by assigning computational tasks to individual threads, thereby reducing overall processing time.

- Serial vs MPI Execution Time Analysis Looking at Figure 1, it's clear that the average execution time for serial bubble sort is 84.31383478 seconds, while employing MPI shows a noticeable decrease to 53.51747778 seconds. This reduction in execution time with an increasing number of processes can be attributed to the efficiency gained by distributing segments of the array to individual processors, subsequently consolidating their outcomes at a centralized location.

- Performance Comparison Overall, OpenMP and MPI implementations provided substantial improvements in execution time compared to the serial implementation. OpenMP, by leveraging shared memory, demonstrated significant speedup by utilizing multiple cores efficiently. MPI, leveraging distributed memory and parallelism, showed competitive performance gains, especially with increased processor count, but incurred communication overhead. The choice between OpenMP and MPI might depend on the system architecture, available resources, and the trade-offs between shared and distributed memory paradigms.

3.3.2 Quick Sort

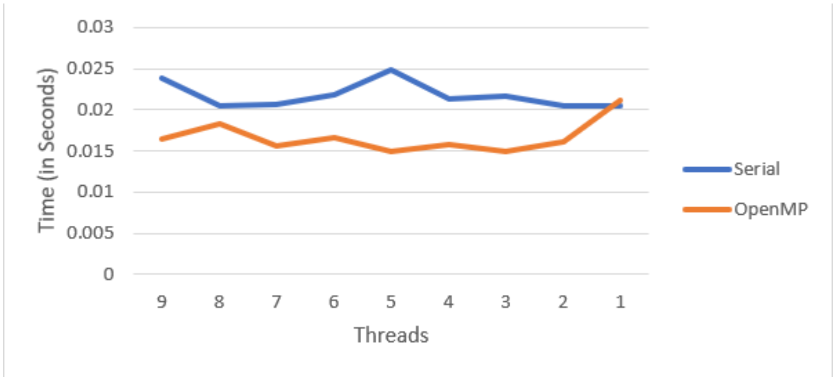

Figure 2: Quick Sort in Serial and OpenMP

– Serial vs OpenMP Execution Time Analysis: Observing fig 2, the average execution time for serial quicksort stands at approximately 0.021 seconds. However, with OpenMP utilization, the average execution time notably decreases to 0.016 seconds. The decrease in execution time signifies the efficiency gained by distributing tasks among threads, resulting in a reduced overall processing duration.

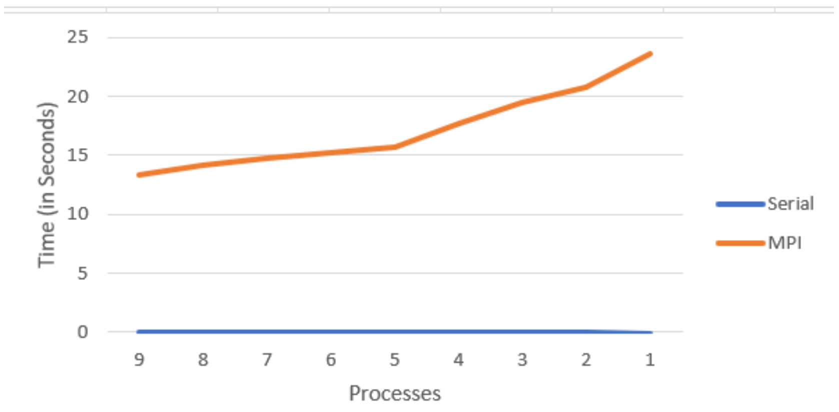

Figure 3: Quick Sort in Serial and MPI

– Serial vs MPI Execution Time Analysis: Observing fig 3, The average execution time for serial quicksort is approximately 0.021 seconds, whereas employing MPI showcases an average execution time of around 16.926 seconds. This decrease in execution time with an increased number of processes can be attributed to the efficiency gained by distributing array segments to individual processors and consolidating their results afterward.

– The choice between OpenMP and MPI might depend on the system architecture, available resources, and the trade-offs between shared and distributed memory paradigms.

3.3.3 Count Sort

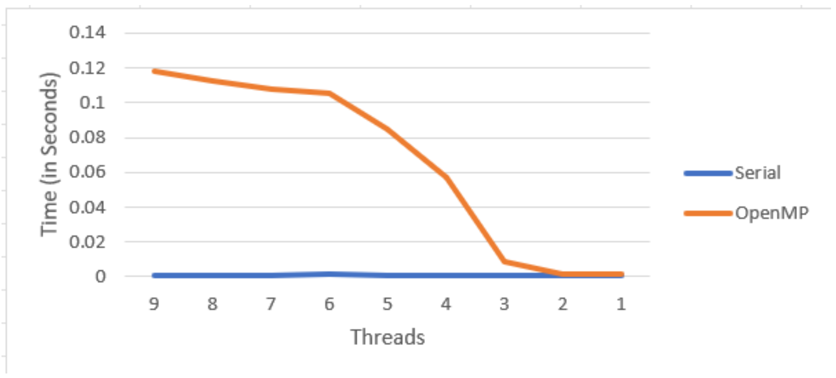

Figure 4: Count Sort in Serial and OpenMP

- Serial vs OpenMP Execution Time Analysis: The average execution time for serial count sort is approximately 0.000951 seconds. In contrast, utilizing OpenMP demonstrates a notable increase in execution time, averaging at 0.047823 seconds. The rise in execution time with increased threads in Figure 4 highlights a significant communication overhead in parallel count sort. This overhead eclipses the parallel efficiency, making the serial count sort a more favorable option due to the added processing time caused by inter-thread communication.

- Serial vs MPI Execution Time Analysis: Serial count sort reports an average execution time of about 0.000951 seconds, whereas employing MPI displays an average execution time of approximately 6.312838 seconds. The increasing execution time as the number of processes rises in Figure 5 underscores a substantial communication overhead in parallel count sort. This overhead becomes more prominent, outweighing the benefits of parallel efficiency and making the serial count sort a preferable choice due to its lack of communication overhead, leading to enhanced processing time.

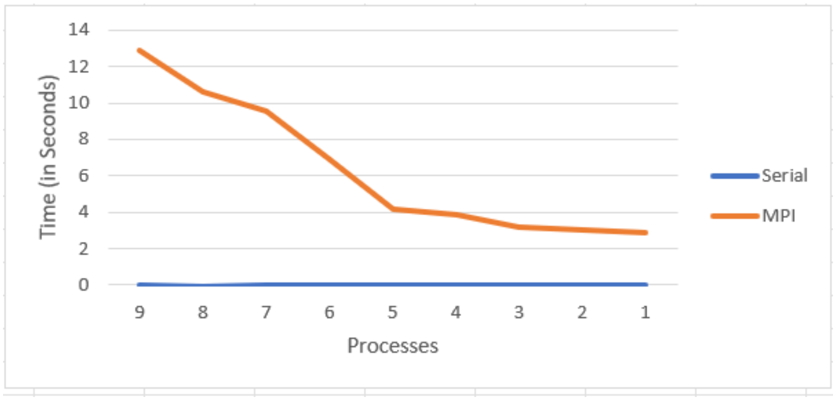

Figure 5: Count Sort in Serial and MPI

– Performance Comparison: The OpenMP and MPI implementation, though aiming for parallelization, experienced increased execution time due to inter-thread communication overhead compared to the serial implementation. Ultimately, despite the parallel capabilities of OpenMP and MPI, the absence of communication overhead in the serial implementation rendered it more efficient in this specific scenario.

4 Conclusion

By evaluating the performance of these sorting algorithms in serial, OpenMP, and MPI paradigms concerning the number of threads/processes and the time taken for sorting specific datasets, this project aims to provide valuable insights into their scalability and efficiency across different computational environments. The analysis will offer guidance on choosing the most suitable sorting algorithm and parallelization approach for various computing scenarios.

Reference

We researched about Bitonic Sort Algorithm from here: **The Implementation and Optimization of Bitonic Sort Algorithm**