CS2006 OPERATING SYSTEMS

PROJECT REPORT

Comparison of Sorting Algorithms between Serial, OpenMP and POSIX

Group members:

- Umar Siddiqui(21K-3197)
- Babar Subzwari(21K-3202)
- Hamza Iqbal(21K-4513)

Objective :

The main focus of this project was to note the performance offered and effort required to implement a program with Serial ,Pthreads and OpenMP, and compare the time taken by three different sorting algorithms to sort sets of random arrays of integers. The three different algorithms integrated in our project are Selection Sort, Merge Sort, and Quick Sort.

This project aims to give insights into the power of parallel programming, as well as which of the two very popular parallel programming approaches(POSIX and OpenMP) is better.

Background :

The C++ programming language is a general-purpose, high-level programming language that was originally developed in the late 1970s. Known for its efficiency, C++ has become one of the most widely used programming languages for system programming, including operating systems development.

C++ provides low-level access to memory and hardware resources, allowing programmers to write efficient and optimized code. It offers a structured approach to programming, supporting functions, data structures, and modular design. The language's syntax is concise and expressive, making it suitable for both small-scale projects and large-scale software development.

In the context of parallel programming, C++ provides a foundation for implementing various parallelization techniques and frameworks. Two widely adopted approaches for parallel programming in C++ are POSIX and OpenMP, with each offering unique features and capabilities.

Pthreads, or POSIX threads, is a standard API for creating and managing threads in a POSIX-compliant operating system. It allows programmers to leverage the parallel processing capabilities of modern systems by creating multiple threads within a single process. Pthreads provides functions for thread creation, synchronization, and communication, enabling developers to exploit parallelism and achieve efficient execution of concurrent tasks.

OpenMP, on the other hand, is an industry-standard API for shared-memory parallel programming. It offers a directive-based approach that allows programmers to add parallelism to their C++ code without significant modifications. OpenMP directives, specified as pragmas, provide hints to the compiler, guiding it to automatically parallelize loops, sections, or even entire programs. OpenMP supports a wide range of parallelization techniques, including parallel loops, task parallelism, and parallel sections.

Serial execution, sometimes referred to as single-threaded execution, represents the traditional sequential approach where tasks are executed one after another on a single processing core. In this mode, computations are carried out sequentially, without exploiting the parallel processing capabilities of modern systems. Serial execution serves as a baseline for evaluating the performance improvements achieved through parallelization techniques like POSIX and OpenMP.

In this project, we aim to compare the performance, scalability, and programming characteristics of POSIX, OpenMP, and serial execution in the context of parallel programming in C++. By evaluating these approaches, we seek to gain insights into their strengths, weaknesses, and trade-offs. The comparison will involve analyzing factors such as execution time, speedup, efficiency, ease of programming, and resource utilization. The findings of this study will help developers make informed decisions about selecting the most suitable parallel programming approach based on their specific requirements and constraints.

Platform and Language :

In this project, we utilized the C++ programming language as the primary language for implementing our parallel programming code. C++ was chosen due to its efficiency, low-level access to system resources, and widespread use in system programming and operating systems development.

The implementation and execution of our code were performed on the Linux Ubuntu operating system. Ubuntu was selected for its popularity, robustness, and extensive support for C++ programming. The Linux environment provided a reliable platform for conducting our experiments and evaluating the performance and characteristics of the parallel programming approaches.

To compile our C++ code, we utilized the G++ (GNU Compiler Collection), which is a widely used compiler for the C++ language. G++ is renowned for its high-quality optimization features, adherence to language standards, and compatibility with various platforms. It provided us with the necessary tools to compile and generate executable binaries from our C++ source code.

The project's development and testing processes were carried out on a computer system equipped with a multi-core processor. The presence of multiple processor cores allowed us to explore and evaluate the parallel execution capabilities of POSIX and OpenMP. By leveraging the parallel processing capabilities of the system, we aimed to measure and compare the performance gains achieved through parallelization against the baseline of serial execution.

The experiments and measurements were conducted multiple times to ensure the accuracy and reliability of the results. We collected data on execution time, speedup, efficiency, and resource utilization, enabling us to evaluate the performance and scalability of the POSIX, OpenMP, and serial execution approaches in different scenarios.

In the subsequent sections of this report, we will describe the methodology in detail, present the results obtained from our project.

Methodology :

Our approach to algorithm comparison was to approach the problem in multiple steps. Firstly we set a hardcoded set of sizes for the program to randomly generate arrays of random numbers for each of those sizes, ranging from One Hundred to to Ten Million. We three functions get times for all three algorithms in an array in all three methods(Serial, OpenMP, POSIX).

The Algorithms used were :
1-) Selection Sort
2-) Quick Sort
3-) Merge Sort

Each Algorithm has a set limit beyond which sorting data took an unreasonable amount of time hence we set limits for the data size of each algorithm. For

Selection Sort it was Ten Thousand variables, For Merge Sort it was One Million Variables and for Quick Sort it was Ten million variables.

After getting the times from all of the methods of each algorithm, the program prints its results in a table format and writes them into a .csv file, the format the data is written in is for Microsoft Excel.

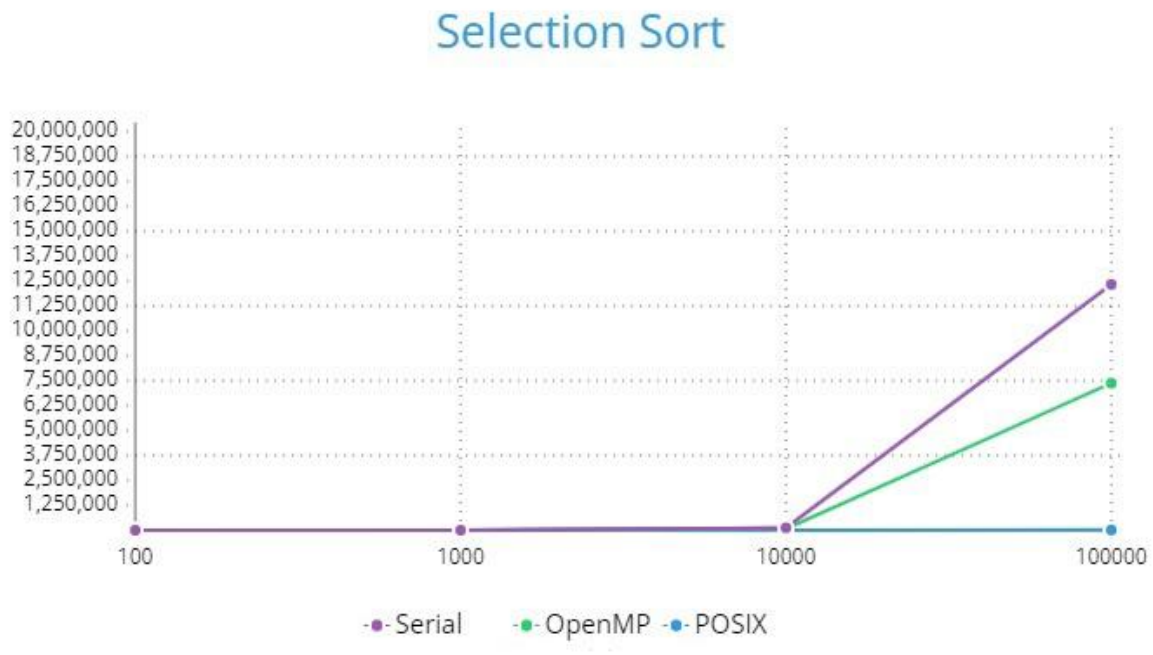
The Methods for Sorting the random variables varied from algorithm to algorithm and the library used.

Selection Sort :

The Serial method is self explanatory and takes the minimum from the array and starts placing them at the start of the array and continues on.

The OpenMP method takes a uses “#pragma omp single” as it is not possible to parallelize the outer loop, meanwhile the inner loop is parallelizable but is simply too inefficient as it ends up taking longer compared to run than a single OpenMP thread.

The POSIX method takes a similar approach though due to POSIX thread functions only taking a single variable a struct was given the thread access to all the needed data.Most of the data division is handled by the thread itself and the speedup is noticeable in larger data sets. POSIX performs much faster compared to the other two methods due to the algorithm not using divide and conquer.



Selection-Sort Times(microseconds)

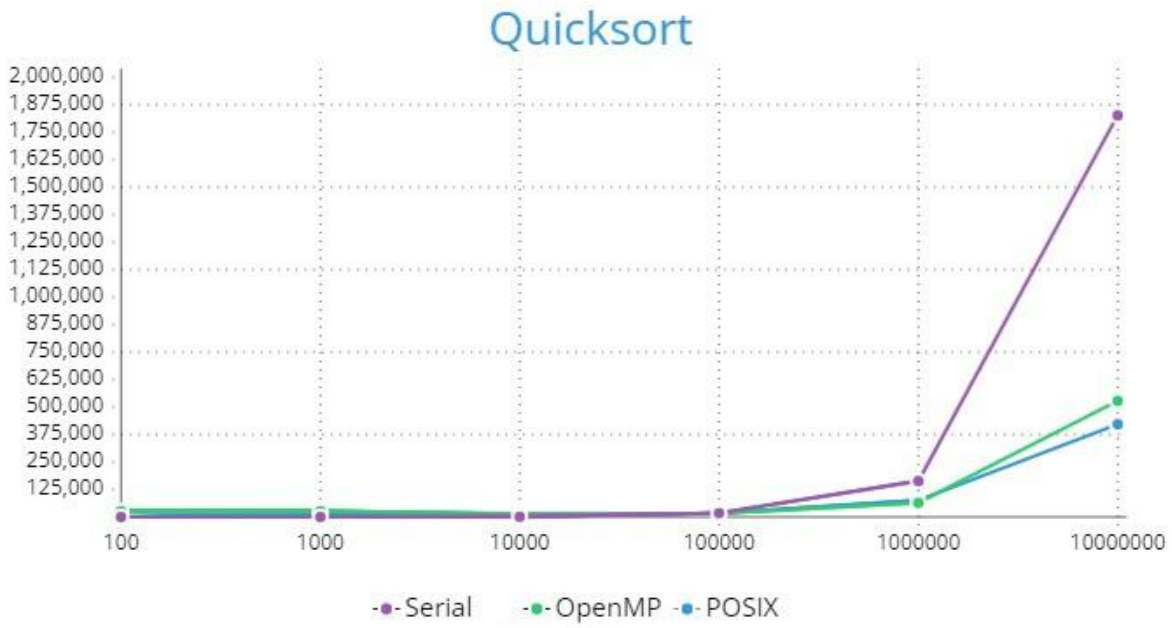
DataSize	Serial	OpenMP	POSIX
100	23	18	571
1000	1895	1171	562
10000	178259	105853	3360
100000	15152881	7569057	7855

Quick Sort:

The Serial version takes the traditional and simple approach with divide and conquer making it faster than the Selection Sort especially in larger datasets and allows it to sort larger sets in general.

The OpenMP method takes the similar recursive approach with each recursion resulting in a new child thread dividing and sorting the array, the number of threads used depends upon the number of threads needed and is set dynamically depending on the size of the dataset. Data sets less than Twelve thousand five hundred are sorted by a two threads each formed by “#pragma omp task” and then sorting serially as excessive thread creation results in time loss due to overhead. The equation to determine the number of threads is at the start of each function, with the limit being decided by a function “lim()” at the start of each of the “Get_xxxxxxx_times()” function.

The POSIX method takes a similar approach to the determining number of threads and also similar to the previous POSIX implementation needs a single variable for its function so in this case an object from a class was used so that the threads may access only their own assigned data. After a certain number of threads have been created they all sort serially after the limit is reached.



Quick-Sort Times(microseconds)

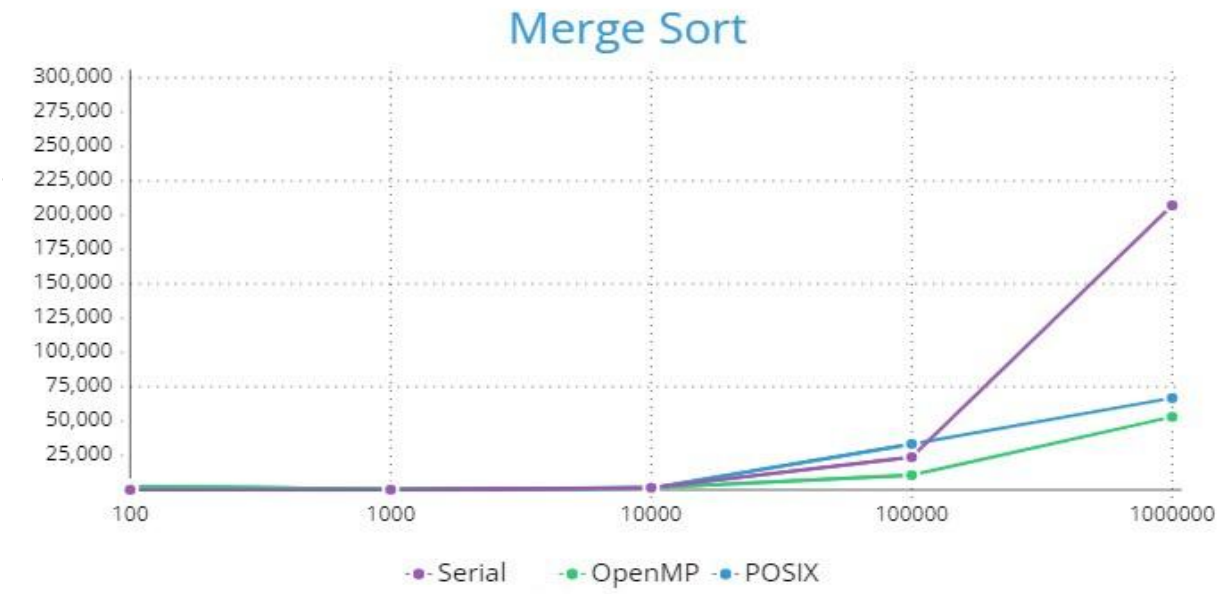
DataSize	Serial	OpenMP	POSIX
100	5	2878	144
1000	88	621	148
10000	1358	1079	1181
100000	18558	12457	20590
1000000	150612	60511	63524
10000000	1828990	469682	628133

Merge Sort :

The Merge Sort implementation is not too much different compared to the Quick Sort as both algorithms apply the rule of divide and conquer. The serial method is as recursive as expected and able to handle larger data sets compared to Selection Sort but less so compared to the Quick Sort, managing to sort up to One million variables.

The OpenMP method takes a similar approach of creating as many threads as needed dynamically dividing arrays and distributing them to each of the threads up to a limit decided by an equation that once approached sets all threads to start sorting Serially as excessive amount of thread creation results in time loss.

The POSIX method is also familiar as it uses the same equations to set a limit for threads created and uses the same class as the one used in the Quick Sort variant of this approach to create objects for each of the threads to assign them all their respective arrays and their supporting indexes.



Merge-Sort Times(microseconds)

DataSize	Serial	OpenMP	POSIX
100	75	5072	1162
1000	150	12194	3033
10000	2423	2109	1836
100000	27941	16527	31219
1000000	212977	59370	105729

Results :

Fully functioning sorting algorithms between Serial,OpenMP and POSIX thread that shows the speed and advantages of each. OpenMP exhibited better performance over Serial programming and Pthreads. OpenMP's high level approach to threading contributed to that it required less effort to implement the algorithms and maintenance of the code was facile due to extensive documentation available. Each Method has its advantages and drawbacks with parallelization resulting in faster clearance times compared to the traditional Serial approaches. OpenMP offers a greater number of options and has different functions depending on what the user needs and even accepts user created datatypes(structs) as inputs in OpenMP 4.0. POSIX on the other hand was more limiting to use as there are less options available which results in fewer ways to solve problems, but also has the advantage of a single reliable implementation. All results are documented into a .csv file within the same folder as the program.

Conclusion :

In conclusion, this report reviews the difference in computation performance between Serial, OpenMP and Pthreads as numerous tests have been performed on parallel execution versus serial execution. Different algorithms were tested with datasets of different sizes generating random values. This gave multiple data points that could be used to show the algorithm performed over a collection of sets in a graph.