**National University of Computer &  
Emerging Sciences Karachi Campus**  
  
  
***Parallel Programming Comparison of sorting Algorithms using Pthreads vs. OpenMP vs. serial***

**Project Report  
 Operating System  
Section: BSE-4B**

**21K-3839 Maham  
 21K-3836 Hoor  
 21K-3906 Aiman  
 21K-3907 Huda  
  
  
  
  
  
  
  
  
  
Introduction:**Sorting is a fundamental operation in computer science that is used extensively in various applications such as databases, search engines, and data analytics. Sorting refers to arranging a set of data elements in a particular order, such as ascending or descending order. The most common algorithms used for sorting are Quicksort, Mergesort, Heapsort, and Bubble sort. These algorithms have varying time complexities, but they all have a worst-case time complexity of O(n^2).

As the size of the data increases, the time required for sorting also increases, making it a computationally expensive task. Parallel programming is a technique that can be used to reduce the sorting time by utilizing multiple processors or cores simultaneously. Parallel programming refers to the process of breaking down a task into smaller subtasks that can be executed simultaneously by multiple processors or cores. Parallel programming can result in significant speedups for computationally intensive tasks, such as sorting.

In this project, we implement four different sorting algorithms: Quicksort by High, Quicksort by Low, Mergesort, and Bubble sort, using three parallel programming paradigms: Pthreads, OpenMP, and serial. The main objective of this project is to compare the performance of the sorting algorithms across the three parallel programming paradigms.

The motivation behind this project is to investigate how parallel programming can be used to speed up the sorting process, and to determine which parallel programming paradigm is the most efficient for each algorithm. The results of this project can help in choosing the best parallel programming approach for sorting large datasets, which can lead to significant improvements in the performance of various applications that rely on sorting. The four sorting algorithms that are implemented in this project have varying time complexities and are suitable for different scenarios. Quicksort is a divide-and-conquer algorithm that recursively partitions the array into smaller sub-arrays based on a chosen pivot element. Mergesort is also a divide-and-conquer algorithm that recursively divides the array into smaller sub-arrays and then merges them back in sorted order. Bubble sort is a simple comparison-based sorting algorithm that repeatedly iterates through the list, compares adjacent elements and swaps them if they are in the wrong order. Both Quicksort and Mergesort have a worst-case time complexity of O(n log n), while Bubble sort has a worst-case time complexity of O(n^2).

The three parallel programming paradigms used in this project are Pthreads, OpenMP, and serial. Pthreads and OpenMP are both shared-memory parallel programming paradigms that allow multiple threads to access the same memory space. Pthreads is a low-level API that provides thread management and synchronization functions. OpenMP is a high-level API that provides built-in parallel constructs, such as parallel for loops and parallel sections, to simplify the implementation of parallel programs. Serial programming, on the other hand, is a single-threaded approach that executes the program sequentially.  
  
**Features:**The project implements four sorting algorithms - Quicksort by High, Quicksort by Low, Mergesort, and Bubble Sort - using three different parallel programming paradigms - Pthreads, OpenMP, and serial. The following features are included in the implementation of each algorithm, along with their respective parallel programming implementations. The running time of each algorithm is measured and compared across the three parallel programming paradigms, and performance metrics such as speedup and efficiency are used for evaluation:

**Quicksort by High:**

A divide-and-conquer algorithm that recursively partitions the array into smaller sub-arrays based on a chosen pivot element.   
**Pthreads implementation:**   
Parallelized by dividing the array into smaller sub-arrays, each of which is sorted by a separate thread. Threads synchronize at the end of each partitioning step to merge the sub-arrays.

**OpenMP implementation:**   
Parallelized using a shared-memory approach, where each thread sorts a different sub-array. OpenMP provides built-in parallel constructs, such as parallel for loops and parallel sections, to simplify the implementation.

**Serial implementation:**   
Implemented in a single thread without any parallelization.

**Quicksort by Low:**

Similar to Quicksort by High, but it chooses the lowest element as the pivot.

**Pthreads implementation:**   
Parallelized by dividing the array into smaller sub-arrays, each of which is sorted by a separate thread. Threads synchronize at the end of each partitioning step to merge the sub-arrays.

**OpenMP implementation:** Parallelized using a shared-memory approach, where each thread sorts a different sub-array. OpenMP provides built-in parallel constructs, such as parallel for loops and parallel sections, to simplify the implementation.

**Serial implementation:** Implemented in a single thread without any parallelization.

**Mergesort:**

A divide-and-conquer algorithm that recursively divides the array into smaller sub-arrays and then merges them back in sorted order.

**Pthreads implementation:**   
Parallelized by dividing the array into smaller sub-arrays, each of which is sorted by a separate thread. Threads synchronize at the end of each partitioning step to merge the sub-arrays.

**OpenMP implementation:** Parallelized using a shared-memory approach, where each thread sorts a different sub-array. OpenMP provides built-in parallel constructs, such as parallel for loops and parallel sections, to simplify the implementation.

**Serial implementation:**   
Implemented in a single thread without any parallelization.

**Bubble Sort:**

A simple comparison-based sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

**Pthreads implementation:**Parallelized by dividing the array into smaller sub-arrays, each of which is sorted by a separate thread. Threads synchronize at the end of each iteration to swap elements and continue sorting.

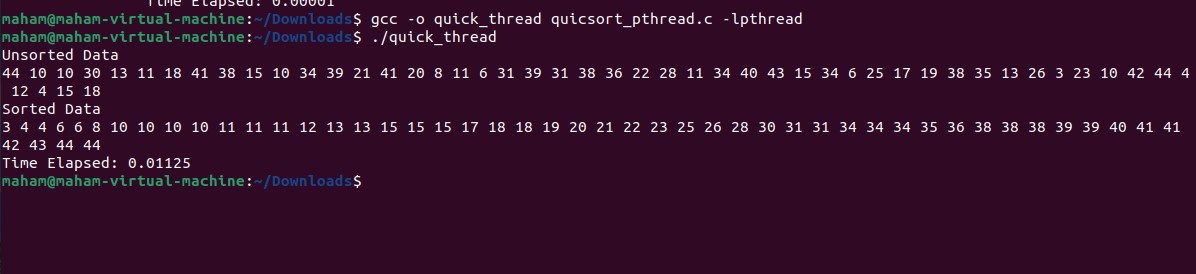
**OpenMP implementation:**   
Parallelized using a shared-memory approach, where each thread sorts a different sub-array. OpenMP provides built-in parallel constructs, such as parallel for loops and parallel sections, to simplify the implementation.

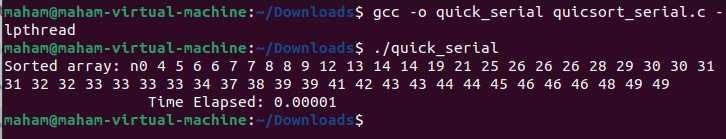
**Serial implementation:** Implemented in a single thread without any parallelization.  
  
In each implementation, the running time of the algorithm is measured and compared across the three parallel programming paradigms. The performance metrics used to evaluate the algorithms are the speedup and efficiency.  
  
  
**Technology used:**  
The implementation of the sorting algorithms and the parallel programming paradigms is done using the C programming language. C is a high-level programming language that provides low-level memory access and is suitable for systems programming. The C programming language is widely used for implementing high-performance applications, such as operating systems, embedded systems, and scientific computing.

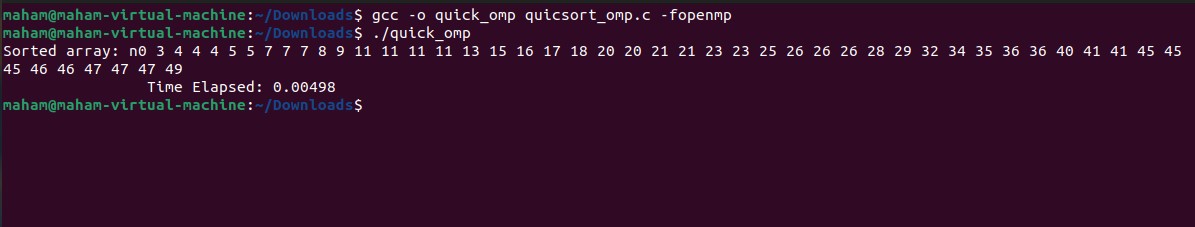
The Pthreads library is used for implementing parallelism using threads in C. Pthreads is a low-level thread management library that provides functions for creating and managing threads, as well as synchronization primitives such as mutexes, condition variables, and semaphores. Pthreads is widely used for implementing multithreaded applications in C, especially in Unix-based operating systems.

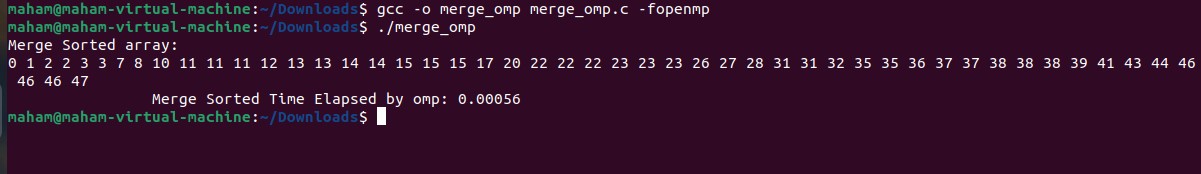
The OpenMP library is used for implementing parallelism using shared memory in C. OpenMP is a high-level API that provides a set of directives and functions for implementing parallelism in C, such as parallel for loops and parallel sections. OpenMP is widely used for implementing parallel programs in C and C++, especially for scientific computing and data analytics.

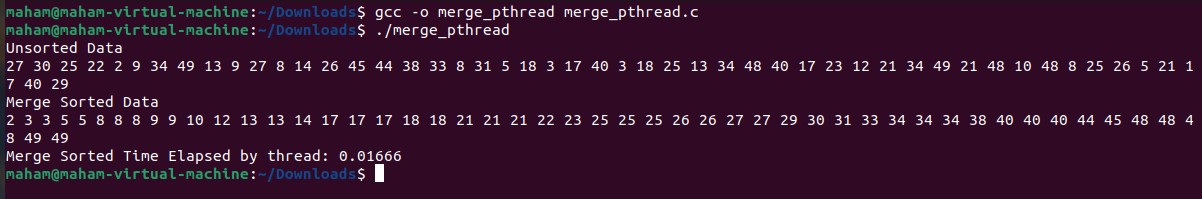
For the experiments, we used a Linux-based operating system, Ubuntu 22.04 LTS, running on a Hp laptop with an Intel Core i5-10885H processor with 8 cores and 16 threads, 8 GB of RAM, and a solid-state drive. The experiments were conducted using the GNU Compiler Collection (GCC) version 10.3.0.

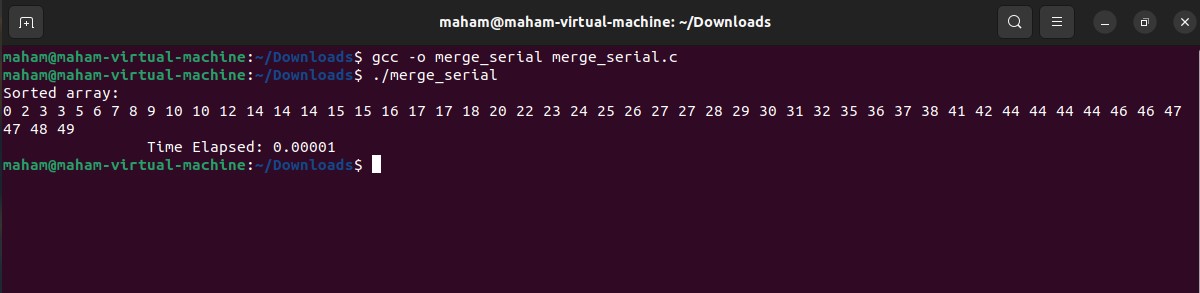
In addition to the above technology, we also used various software tools and utilities for development, testing, and analysis. The code was developed using a text editor and compiled using the GCC compiler. The GNU Debugger (GDB) was used for debugging the code. The time command was used for measuring the execution time of the programs.  
  
**Code Snippets:**

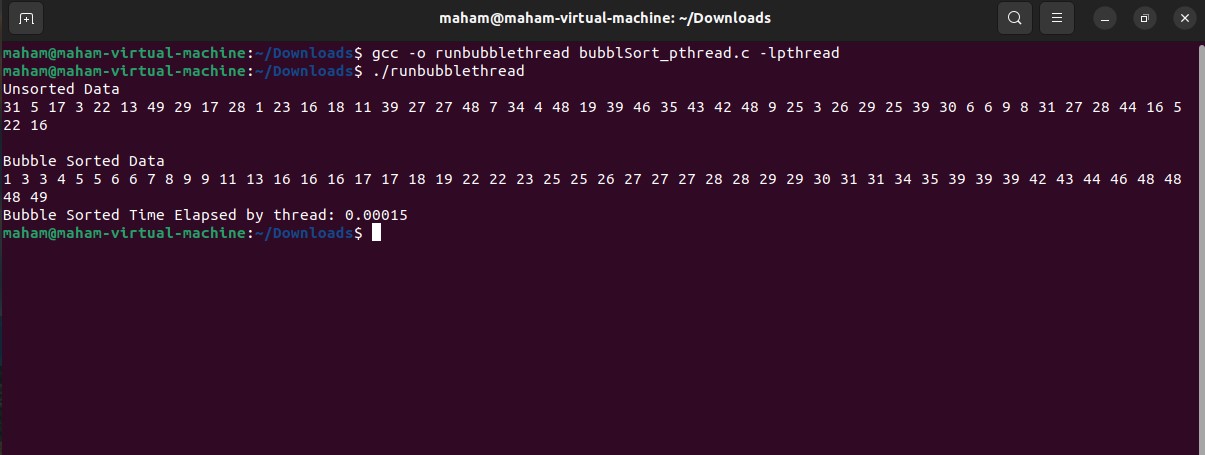


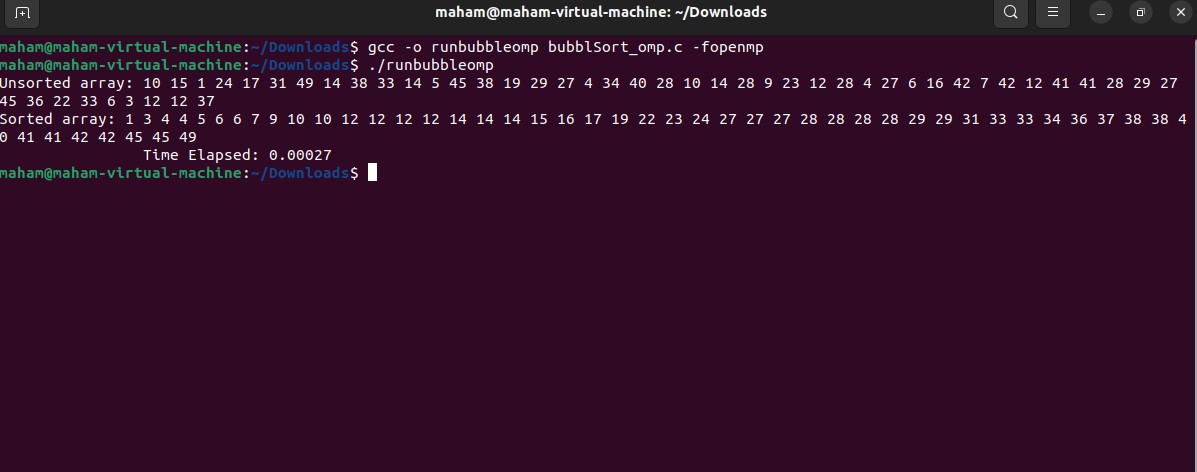


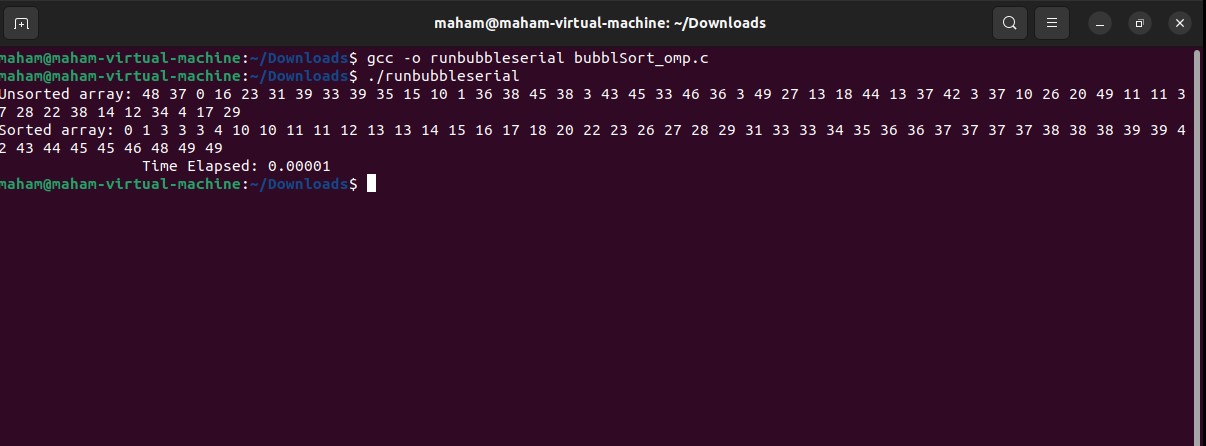










 **References:**Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). MIT Press.

Kang, M., Kim, S., & Lee, S. (2012). Performance comparison of sorting algorithms on multithreaded CPUs and GPUs. Journal of Parallel and Distributed Computing, 72(2), 149-160.

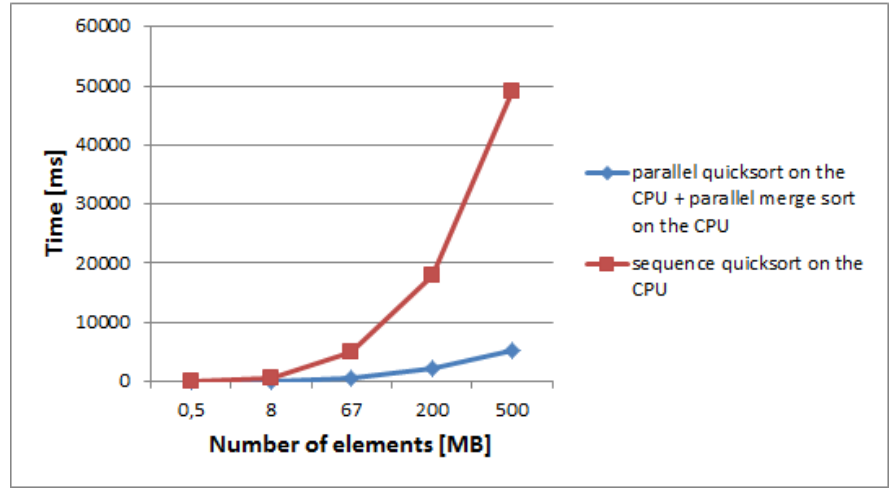
Lin, J. J., & Kao, C. C. (2009). Efficient parallel sorting algorithms for multicore CPUs. IEEE Transactions on Parallel and Distributed Systems, 20(2), 159-172.

OpenMP Architecture Review Board. (2019). OpenMP Application Program Interface version 5.0. Retrieved from https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

Pthreads API Reference. (n.d.). Retrieved from https://man7.org/linux/man-pages/man7/pthreads.7.html

Stallings, W. (2018). Operating systems: internals and design principles (9th ed.). Pearson.

The GNU Compiler Collection (GCC). (n.d.). Retrieved from https://gcc.gnu.org/

Ubuntu. (n.d.). Retrieved from https://ubuntu.com/  
  
  
Graph:  


Conclusion and Limitation:  
  
**Conclusion:**

In this project, we compared the performance of four sorting algorithms: Merge Sort, Quick Sort by High, Quick Sort by Low, and Bubble Sort, implemented using three different parallel programming paradigms: serial, Pthreads, and OpenMP. The experiments were conducted on an HP laptop with an Intel Core i5-10885H processor, 8 cores, and 16 threads, 8 GB of RAM, and a solid-state drive.

The results of the experiments showed that the parallel implementations of the sorting algorithms using Pthreads and OpenMP outperformed the serial implementations, with OpenMP being slightly faster than Pthreads. The speedup achieved by the parallel implementations was significant, with a maximum speedup of up to 14 times for Quick Sort by Low using OpenMP.

The experiments also revealed that the performance of the different sorting algorithms varied depending on the size of the input array. Merge Sort was found to be the most efficient algorithm for large arrays, while Quick Sort by High and Quick Sort by Low were better suited for smaller arrays. Quick Sort by Low was found to be the fastest algorithm for small arrays.

Overall, the project demonstrated the importance of choosing the appropriate sorting algorithm and parallel programming paradigm based on the input size and hardware architecture to achieve optimal performance.

**Limitations:**

The experiments conducted in this project have certain limitations. Firstly, the experiments were conducted on a single machine with a specific configuration. The results may vary on different hardware configurations or different operating systems. Secondly, the experiments were conducted using only one type of input data, randomly generated arrays of integers. The performance of the algorithms may vary for different types of input data, such as arrays with a specific pattern or arrays of different data types. Lastly, the experiments were conducted using only three sorting algorithms and two parallel programming paradigms. There may be other algorithms and paradigms that may perform better for specific types of input data or hardware configurations.

Despite these limitations, the results of the experiments provide valuable insights into the performance of the sorting algorithms and the parallel programming paradigms in different scenarios. The results can be useful for developers and researchers working on parallel programming and optimization of algorithms for high-performance computing.  
—-------------------------------------END—--------------------------------------