**Performance Comparison of Sorting Algorithms**

**Group Members**

Khuzaima Ahsan 21K-3328

Aahil Ashiq Ali 21K-4549

Khubaib Khan Lodhi 21K-4596



**Teacher Name: Sir Nadeem Kafi Khan**

# OPEN MP:

# Introduction

In this project, we implemented parallel programming on sorting random numbers using three different sorting algorithms and compared their performance. The performance of each algorithm is subjective of the time taken to sort the data. We tested count sort, merge sort and selection sort algorithms. We changed the input size on all 3 sorting algorithms and displayed the time taken on single thread and multiple threads.

# COUNT SORT

Counting sort is an algorithm for sorting a collection of objects according to keys.

This code can help in performance analysis of merge sort algorithm and trends from small dataset of 1024 to the length of data set the user provides.

The counting sort algorithm does a better job when the range of the n items is within n. However, when the range is increased to **n2**, the performance of counting sort is worse than the comparison-based sorting algorithms. Either you can use radix sort to solve the problem or parallelize the existing code by not making count array but instead determining the index of every element in count variable and placing in output array.

# 1M

# 

But even this count variable approach cannot solve this problem for a data as huge as 1 B.

Here, the outer loop has been parallelized which can produce resource contestation and incorrect sorted arrays for very larger arrays. Instead, you can use whole count arrays and first find indexing and then place it but this would come at the cost of higher time complexity. It is not in place sorting. But you can do so again by using count arrays instead of variable, but time complexity would increase.

# MERGE SORT

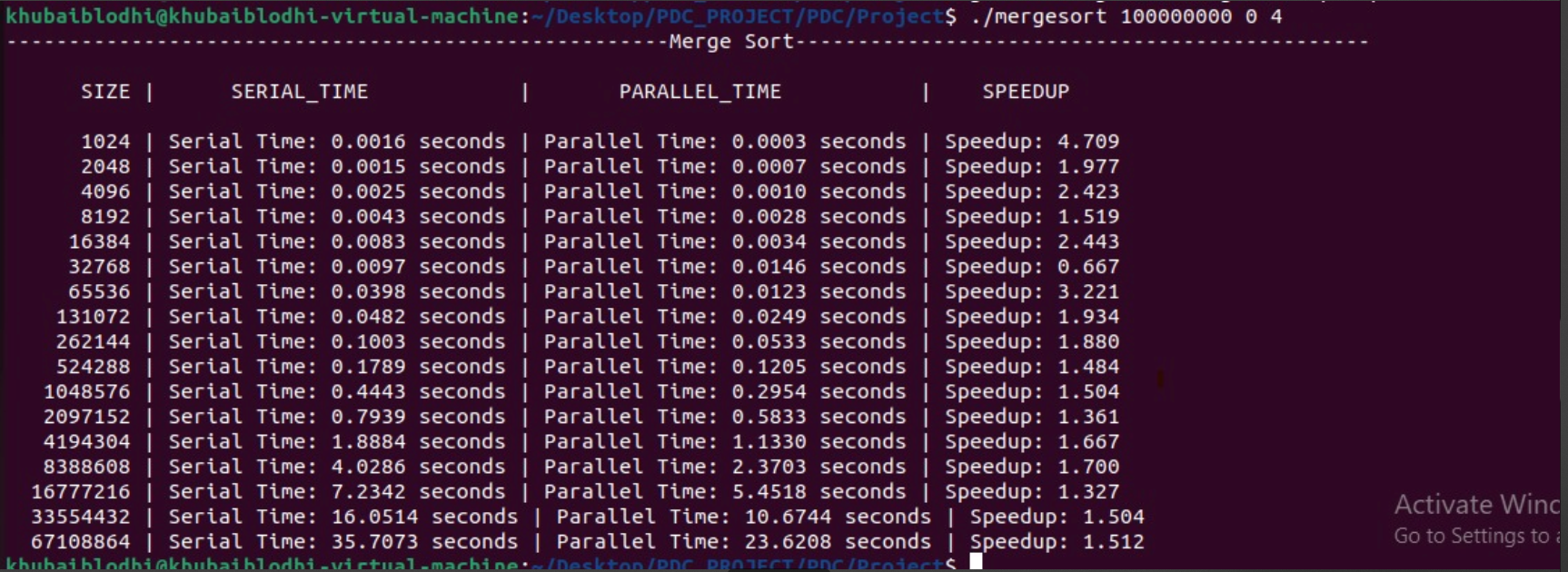
# Merge sort is defined as a [sorting algorithm](https://www.geeksforgeeks.org/sorting-algorithms/) that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

This code can help in performance analysis of merge sort algorithm and trends from small dataset of 1024 to the length of data set the user provides.

**Parallelization Strategy:**

* OpenMP is used to parallelize the merge sort algorithm.
* The parallelization is applied to the outer loop of the merge sort, where the array is divided into segments and each segment is sorted independently. The merging step is performed sequentially.
* Scheduling is done dynamically

# 100 M



In the merge function, we have preferred dynamic memory allocation for arrays because of large data sets, it could create issues with stack size. But if you have the capacity to increase stack size you should use stack based local arrays. You can increase stack size given to thread by command OMP\_STACKSIZE=2048M. We have used the iterative method rather than recursive in merge Sort function because of overhead problems. We are scheduling the code dynamically which is giving us like speed up of 150%

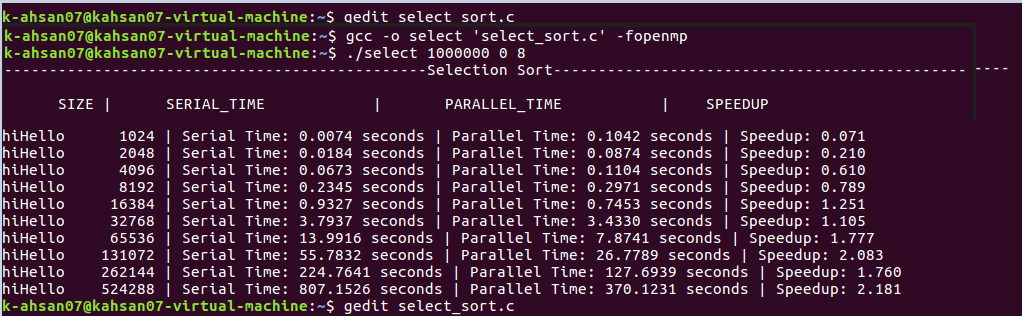
# Selection Sort

Selection sort is a simple comparison-based sorting algorithm with a time complexity of O(n^2). The parallelization strategy involves distributing the sorting workload among multiple threads to exploit parallel processing capabilities.

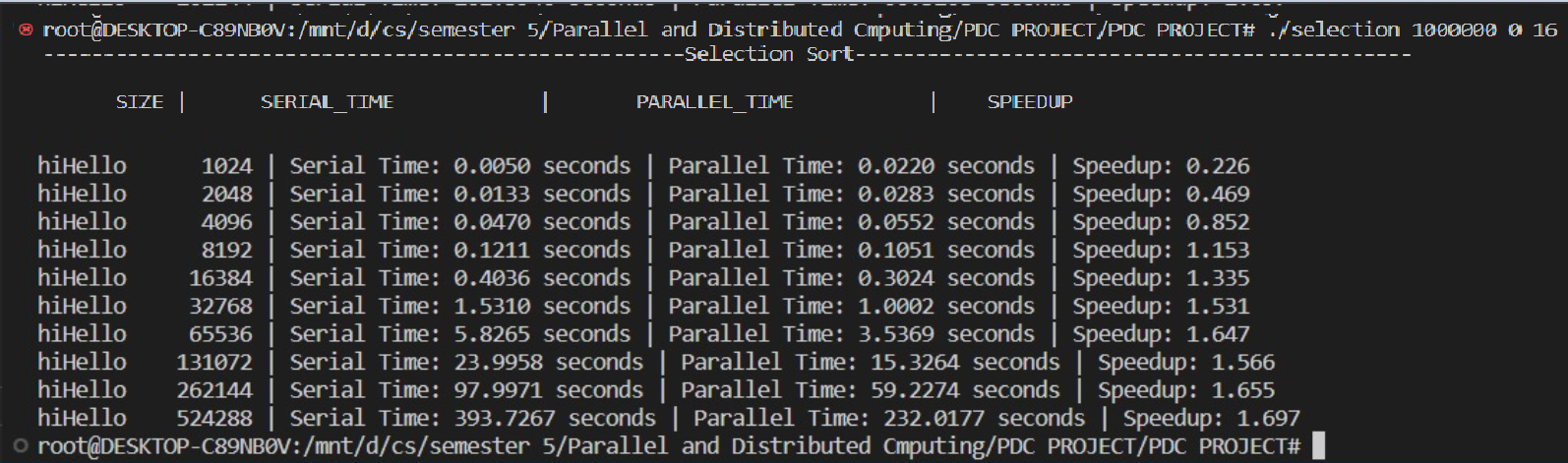
Parallelization Approach:

* The code parallelizes the selection sort algorithm by dividing the array into chunks for parallel processing.
* Each thread independently determines the maximum element within its assigned range, and a critical section is used to update the global maximum.
* Load balancing is achieved through OpenMP directives, ensuring that each thread contributes to the sorting process, minimizing idle time and maximizing CPU utilization.
* The custom reduction operation improves the efficiency of finding the maximum element and its index in each thread.

# 1M – 8 Threads



# 1M – 16 Threads



It shows that with the increase of threads, the speedup is less. This is possibly due to high resource contentions and data dependencies.

# MPI:

For MPI, processes are made which are based on concept of distinct distributed memory. MPI is particularly useful in algorithms where there are less data dependencies and sharing. Normally merge sort is used for parallelizing into MPI because of already optimized data and less dependencies in merge sort function. But merge sort parallelization using mpi is far more complex because of significant communication needed in merge function. So we parallelized selection sort giving very high speedups.

# Selection Sort

Selection sort is inherently sequential sort. Selection sort has a fixed sequence of steps, and the selection of an element in one iteration depends on the results of the previous iterations. Therefore, while there are no data dependencies within a single iteration, the algorithm has a sequential nature. Each iteration depends on the results of the previous one to determine which element to select next.

But still, parallelizing it gives significant boost to time efficiency.

# 100K

# 

# Speed up = 5.38

# 1M

# 

# Speed up with 4 processes= 2.5 Speed up with 64 processes = 46.375

# It shows that increasing the no of processes increases the speed manifold.