



# ***Parallel Quicksort Algorithms: A Comparative Study***

Introduction and Literature Review

Presented By:

Saad Ahmed Khan 24440

Beena Ahmed 25240

Beenish Ahmed 25147

Course : Parallel and Distributed Computing  
Instructor: Sir Zainuddin

## **Table of Contents**

1. Introduction
2. Objectives
3. Sequential Quick Sort Algorithm
4. Complexity Analysis:
5. Exploring Parallel Quick Sort Algorithms: A Comprehensive Literature Review
  - 5.1 Abstract
  - 5.2 Reason for parallelizing Quick Sort
6. Approaches to Parallel QuickSort
  - 6.1 Parallel Programming Libraries/APIs:
  - 6.2 Parallel Sorting Algorithms
  - 6.3 Combination of Sequential and Parallel Approaches
7. Analysis of Parallel Quicksort Algorithms:
  - 7.1 Time and space complexity analysis
  - 7.2 Scalability and load balancing
  - 7.3 Performance evaluation
8. Conclusion

## 1.Introduction

Computer science relies heavily on sorting algorithms, and several algorithms, including Quicksort, Merge sort, Bubble sort, Insertion sort, and Selection sort, have been evolved. In particular, Quicksort is well-researched and renowned for its effectiveness. By dividing the sorting problem into smaller subproblems and resolving each one, it employs the "divide and conquer" tactic. For huge datasets, Quicksort is frequently chosen over other sorting algorithms because of its strong performance in terms of average-case time complexity.

A possible way to get around the drawbacks of sequential algorithms like Quick Sort is to use parallel processing. Sorting algorithm performance can be greatly enhanced by parallel processing, which splits jobs into smaller, independent units that can be run concurrently.

This project aims to explore various parallel Quick Sort algorithms, their implementations, and their performance characteristics.

***(Different Quicksort Algorithms,2020)***

## 2.Objectives

- To gain a deeper understanding of the fundamentals of parallel programming and the QuickSort algorithm.
- Develop a comprehensive understanding of parallel programming and the Quick Sort algorithm.
- To apply and evaluate various parallel Quick Sort algorithms and libraries , such as MPI, POS IX, Openmp,Parallel Quicksort by Regular Sampling (PRSR), Optimized Parallel Quick Sort, Hyperquicksort,
- Assess and contrast the performance of these algorithms based on time and space complexity.
- Identify the advantages and disadvantages of each algorithm, and determine their suitability for various use cases.

The methodology of this project involves a comprehensive literature review, algorithm implementation, and performance analysis. The literature review will focus on understanding the theoretical aspects of parallel programming and Quick Sort, as

well as the existing parallel Quick Sort algorithms. The implementation phase will involve coding these algorithms in a suitable programming. Finally, the performance analysis will compare the algorithms based on metrics such as execution time, speedup, and efficiency.

In the following sections, we will delve into the details of each parallel Quick Sort algorithm, their implementations, and their performance characteristics.

### **3. Sequential Quick Sort Algorithm**

Quick sort starts by selecting a pivot element from the list and partitioning the other elements around it. The elements smaller than the pivot are moved to its left, while the larger elements are moved to its right. This process is recursively applied to the sublists until the entire list is sorted.

#### **Algorithm Steps:**

1. Find a random pivot  $p$ .
2. Partition the list in accordance with this pivot, elements less than pivot to the left of pivot, elements greater than pivot to the right of pivot and elements equal to pivot in the middle.  $<p = p > p$ .
  - That is, initialize  $i$  to the first element in the list and  $j$  to the last element.
  - Increment  $i$  until  $\text{list}[i] > \text{pivot}$
  - Decrement  $j$  until  $\text{list}[j] < \text{pivot}$
  - Repeat the above steps until  $i > j$ .
  - Replace pivot element with  $\text{list}[j]$
3. Recurse on each partition.

When the list size is 1, it terminates. This acts as the base case. At this point the partitions are in sorted order so it merges them forming a complete sorted list..

**(Parallel Quick Sort, 2021)**

#### 4.Complexity Analysis:

The complexity analysis of Sequential Quicksort considers the best case, average case, and worst case scenarios. The best case occurs when the input is evenly divided into subproblems, while the worst case happens when the pivot element consistently creates unbalanced partitions.

The time complexity is  $O(n \log n)$  in the average case. The space complexity is  $O(\log n)$ . However the time and space complexity in the worst case is  $O(n^2)$ . (**Parallel Quick Sort, 2021**)

# Literature Review

## 5. Exploring Parallel Quick Sort Algorithms: A Comprehensive Literature Review

### 5.1 Abstract:

Sorting is a key part of computer science, and Quick Sort works well for most tasks. As parallel computing becomes more common, it's important to update Quick Sort to use multiple processors. This review looks at different ways to do parallel Quick Sort, how they work, and how well they perform. It also talks about why we need parallel Quick Sort and the challenges and future directions in this area.

(M.Broussard, 2024)

### 5.2 Motivation for parallelizing Quick Sort

Sequential Quicksort is a great sorting algorithm, but it has some problems because it works linearly.. This means it can't take full advantage of modern computers with multiple processors. When dealing with large amounts of data, this can slow things down, and in the worst-case scenario, it can take a lot of time ( $O(n^2)$ ).

Parallel Quicksort solves these issues by distributing sorting into smaller parts that can be done at the same time by different processors. This makes parallel quick sort great for larger arrays..

(Heidelberger et al., 1990)

## 6. Approaches to Parallel Quick Sort:

### 6.1. Parallel Programming Libraries/APIs:

- OpenMP (omp.h): A shared-memory parallel programming API for C, C++, and Fortran.
- MPI: A message-passing library for distributed-memory parallel programming.
- POSIX threads (Pthreads): A threading library for C and C++ that allows the creation and management of multiple threads within a single process.

### **1. OpenMP:**

OpenMP is a widely-used API for shared-memory parallel programming in C, C++, and Fortran. It provides a set of directives, functions, and environment variables to specify and control parallelism in a program. OpenMP is suitable for multi-core systems or SMPs, where threads can share the same memory space. OpenMP supports both task-based and loop-based parallelism, making it flexible and easy to use for a wide range of applications.

### **2. PTHREAD:**

POSIX threads (Pthreads): Pthreads is a threading library for C and C++ that allows the creation and management of multiple threads within a single process. Threads share the same memory space, making it efficient for tasks with frequent data exchange. Pthreads is suitable for shared-memory parallel programming on multi-core systems or symmetric multiprocessors (SMPs).

### **3. MPI:**

.MPI stands for Message Passing Interface. It is a standardized and portable message-passing system designed for parallel computing on distributed-memory architectures, such as clusters of computers or massively parallel processors (MPPs). MPI provides a set of functions and libraries that enable communication and data exchange between processes executing on different nodes in a distributed-memory system

## **6.2 Parallel Sorting Algorithms**

### **1. Optimized Parallel Quick Sort**

Optimized Parallel Quicksort is an optimized version of the previously mentioned technique. It uses techniques like optimized pivot selection, load balancing, and minimizing communication overhead to increase efficiency.

In this method, we make a small change to how many processes we use at each step. Instead of doubling the number of processes each time, we use the same  $n$  number of processes throughout the entire algorithm. These processes all work at the same time at each step to find the pivot and rearrange the list.

### **Steps.**

1. Start  $n$  processes where each process partitions the array and sorts the sub array using a chosen pivot.
2. Each of the partitions work on their own sub array.
3. Each process finds its own pivot and divides the list based on that pivot.
4. Finally all the sorted sub arrays are merged into one array(**Mondal, 2022**)

## **2. Hyperquicksort**

Hyper Quicksort is another type of quicksort which is used to sort an array in ascending or descending order. It uses the divide and conquer strategy. On an average it takes  $O(n \log n)$  complexity, making it suitable for sorting huge data volumes.(**Miller,2021**)This variant of the quicksort is efficient as it makes it more likely to find a median by sorting the sub arrays sequentially using one pivot that is shared amongst to all processes at the beginning of the algorithm.

### **Steps:**

1. An array of size  $n$  is divided amongst  $n$  processes. So for example if we have an array of size 16 and we have 4 processes, then each process will receive and process 4 elements.
2. A process among the four responsible for finding the pivot element, finds a pivot and broadcasts it to all processes which sort their sublists sequentially using the



broadcasted pivot element. This step will improve chances of finding pivots close to the true median.

3. A process first finds a pivot element in its sub array. Then this process shares its pivot to all other processes. Then each process sorts their own sub array sequentially using the shared pivot. This helps find pivots that are likely to be close to the median of the sub array
4. Repeat steps 4-6 from the previous approach,
  - Pivot selection and broadcasting to partner processes.
  - Dividing the array into subarrays.
  - Swapping of values between partner processes.
5. The remaining top half from one process and the top half from the other partner process are merged into sub array for each process.
6. Traverse the upper half and lower half of each subprocess to achieve a sorted list.
7. Finally merge the sorted sub arrays from all the processes in order to get a fully sorted list. **(Mondal, 2022)**

### **3. Parallel Quick Sort by Regular Sampling**

Parallel Quicksort by Regular Sampling is an algorithm that is suitable for a diverse range of MIMD architectures. **(Li et al., 1993)** It utilizes sampling techniques to efficiently partition the input array and distribute workload across processors while maintaining load balance and reducing communication overhead.

#### **Steps:**

1. The input array is divided amongst  $n$  processes.
2. Each process sorts its received sub array via the sequential quicksort algorithm.

3. All processes select regular samples from their sorted sub arrays.
4. A single process is selected and that process gathers the samples, sorts them and then shares the processed selected pivots to other processes.
5. Each process uses selected pivots to divide their sub array into further sub arrays according to selected pivots one by one.
6. After this processes communicate with one another to swap sorted values with other partner processes.
7. The sorted sub arrays in individual processes are merged to get a fully sorted list.(Li et al., 1993)

### **6.3 Combination of sequential and parallel approaches**

This approach combines elements of sequential and parallel processing to achieve optimal performance, leveraging sequential Quick Sort for small sub-arrays and transitioning to parallel execution for larger ones to exploit parallelism effectively. .  
(*Parallel Quick Sort*, 2021)

## **7. Analysis of Parallel Quick Sort Algorithms:**

### **7.1 Time and space complexity analysis**

According to several empirical evidence, parallel quicksort algorithms are typically faster than the standard sequential quick sort algorithm by a factor of 2 or sometimes even more. Parallel quicksort algorithms are not only better in terms of time complexity but they also utilize fewer computer resources so they are better in terms of space complexity aswell. (Hossain et al., 2020; Xiang, 2011)

### **7.2 Scalability and load balancing**

Efficiency of parallel quick sort algorithms also improve alot when we increase computer resources such as memory or cores, which was not really the case with sequential quick sort(Shah, 2024)

## **Conclusion**

To conclude, our literature review explains the importance of parallelizing algorithms by proving how parallel quicksort algorithms are much better in terms of efficiency compared to sequential quicksort. Various kinds of techniques such as hyper quick sort and random sampling to parallelize quicksort were explained in this report.

## References

[1] *Different Quicksort Algorithms*

<https://sscoetjalgaon.ac.in/public/pdfs/ijssbt/IJSSBT7-2.pdf#page=6>

[2] *Parallel Quick Sort.* (2021, November 14). OpenGenus IQ: Computing Expertise & Legacy.

<https://iq.opengenus.org/parallel-quicksort/>

[3] *Parallel Quick Sort.* (2021, November 14). OpenGenus IQ: Computing Expertise & Legacy.

<https://iq.opengenus.org/parallel-quicksort/>

[4] SORTING ALGORITHMS: QUICK SORT

<https://www.linkedin.com/pulse/sorting-algorithms-quick-sort-ashley-m-broussard-flbhe/>

[5] Heidelberg, P., Norton, A., & Robinson, J. T. (1990). Parallel Quicksort using

fetch-and-add. *IEEE Transactions on Computers*, 39(1), 133–138.

<https://doi.org/10.1109/12.46289>

[6] Mondal, R. (2022, December 7). Parallel Quick Sort. *Medium*.

<https://rm9817696700.medium.com/parallel-quick-sort-7f2240e92128>

[7] HYPER QUICK SORT

<https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Mrunal-Narendra-Inge-Spring-202>

- [8] Mondal, R. (2022, December 7). Parallel Quick Sort. *Medium*.  
<https://rm9817696700.medium.com/parallel-quick-sort-7f2240e92128>
- [9] Li, X., Lu, P., Schaeffer, J., Shillington, J., Pok Sze Wong, & Shi, H. (1993). On the versatility of parallel sorting by regular sampling. *Parallel Computing*, 19(10), 1079–1103.  
[https://doi.org/10.1016/0167-8191\(93\)90019-H](https://doi.org/10.1016/0167-8191(93)90019-H)
- [10] Li, X., Lu, P., Schaeffer, J., Shillington, J., Pok Sze Wong, & Shi, H. (1993). On the versatility of parallel sorting by regular sampling. *Parallel Computing*, 19(10), 1079–1103.  
[https://doi.org/10.1016/0167-8191\(93\)90019-H](https://doi.org/10.1016/0167-8191(93)90019-H)
- [11] *Parallel Quick Sort*. (2021, November 14). OpenGenus IQ: Computing Expertise & Legacy.  
<https://iq.opengenus.org/parallel-quicksort/>
- [12] Hossain, Md. S., Mondal, S., Ali, R. S., & Hasan, M. (2020). Optimizing Complexity of Quick Sort. In N. Chaubey, S. Parikh, & K. Amin (Eds.), *Computing Science, Communication and Security* (pp. 329–339). Springer.  
[https://doi.org/10.1007/978-981-15-6648-6\\_26](https://doi.org/10.1007/978-981-15-6648-6_26)
- [13] Xiang, W. (2011). Analysis of the Time Complexity of Quick Sort Algorithm. 2011 *International Conference on Information Management, Innovation Management and Industrial Engineering*, 1, 408–410.  
<https://doi.org/10.1109/ICIII.2011.104>
- [14] Hyper Quick Sort/ Parallel Quick Sort  
<https://medium.com/@jwalitshah2q/hyper-quick-sort-a16eee2e4093>
- [15] Nyholm, J. (n.d.). *A study about differences in performance with parallel and sequential sorting algorithms*.  
<https://www.diva-portal.org/smash/get/diva2:1559456/FULLTEXT01.pdf>

# Project Task 3 : Code Submission

## Introduction

The following C programs compare the performance of four different sorting algorithms: **sequential quicksort, parallel quicksort, hyperquicksort and Parallel Quick Sort by Regular Sampling** and 3 Parallel Programming Libraries: **open mp, mpi and Posix**. These algorithms are then tested with arrays of different sizes to compare their time performances.

### **1st Program - Code for sequential quicksort, parallel quicksort, hyperquicksort :**

The program first initializes three arrays of size N (which is defined as 1,000,000) with random integers. It then sorts each array using one of the three sorting algorithms and measures the time taken by each algorithm using the `omp_get_wtime()` function from the OpenMP library.

#### **Functions :**

```
quickSort()
```

This function is a standard implementation of the quicksort algorithm, which is a divide-and-conquer algorithm that recursively sorts sub-arrays.

```
parallelQuickSort()
```

This function is a parallelized version of the quicksort algorithm using OpenMP tasks. It is similar to the sequential version, but it creates two new tasks to sort the two sub-arrays instead of recursively calling the function.

```
hyperquicksort()
```

This function is an implementation of the hypercube-based quicksort algorithm, which is designed to work on a distributed-memory parallel computer with a hypercube network topology. The function first sorts each node's sub-array using the sequential quicksort algorithm. It then iteratively partitions and merges the sub-arrays based on the median key of the nodes with a relative node number of 0. The function uses OpenMP parallel for loops to sort and merge the sub-arrays in parallel.

```
main()
```

It initializes three arrays Array, Array1, and Array2 with random integers.  
Measures the time taken for sequential quicksort, parallel quicksort, and hyperquicksort.  
Prints out the elapsed time for each sorting algorithm.

### Output :

The output of the program shows the time taken by each sorting algorithm to sort an array of size N. The results show that the **parallel quicksort algorithm is the fastest, followed by the sequential quicksort algorithm**, and the **hyperquicksort algorithm is the slowest**. This is likely because the hyperquicksort algorithm is designed for a distributed-memory parallel computer with a specific network topology, and it may not be well-suited for a shared-memory parallel computer or a single-threaded execution.

### Code :

```
#include <stdio.h>
#include <omp.h>
```

```
#define N 1000000 // Size of the array
#define d 4 // Number of dimensions in the hypercube network
```

```
void quickSort(int Array[], int first, int last);
void parallelQuickSort(int Array[], int first, int last);
void hyperquicksort(int Array[], int node_number);
```

```
int main() {
    int Array[N];
    int Array1[N];
    int Array2[N];
```

```

// Seed the random number generator
srand(123); // Use a constant seed for reproducibility

// Initialize the array with random integers
for (int i = 0; i < N; i++) {
    Array[i] = rand() % 100; // Generates random numbers between 0 and 99
    Array1[i] = rand() % 100; // Generates random numbers between 0 and 99
    Array2[i] = rand() % 100; // Generates random numbers between 0 and 99
}

// printf("Unsorted Array: ");
// for (int i = 0; i < N; i++) {
//     printf("%d ", Array[i]);
// }
// printf("\n");

// Measure time for sequential quickSort
double start_time = omp_get_wtime();
quickSort(Array, 0, N - 1);
double end_time = omp_get_wtime();
printf("Sequential Quicksort elapsed time: %.6f seconds\n", end_time - start_time);

// Measure time for parallelQuickSort
start_time = omp_get_wtime();
#pragma omp parallel
{
    #pragma omp single
    {
        parallelQuickSort(Array1, 0, N - 1);
    }
}
end_time = omp_get_wtime();
printf("Parallel Quicksort elapsed time: %.6f seconds\n", end_time - start_time);

start_time = omp_get_wtime();
hyperquicksort(Array2, 0);
end_time = omp_get_wtime();

printf("Hyper Quicksort elapsed time: %.6f seconds\n", end_time - start_time);

// printf("Sorted Array: ");
// for (int i = 0; i < N; i++) {

```



```

    // printf("%d ", Array2[i]);
    // }
    // printf("\n");

    return 0;
}

void quickSort(int Array[], int first, int last) {
    int pivot, j, temp, i;
    if (first < last) {
        pivot = first;
        i = first;
        j = last;
        while (i < j) {
            while (Array[i] <= Array[pivot] && i < last)
                i++;
            while (Array[j] > Array[pivot])
                j--;
            if (i < j) {
                temp = Array[i];
                Array[i] = Array[j];
                Array[j] = temp;
            }
        }
        temp = Array[pivot];
        Array[pivot] = Array[j];
        Array[j] = temp;

        // Recursive calls
        quickSort(Array, first, j - 1);
        quickSort(Array, j + 1, last);
    }
}

void parallelQuickSort(int Array[], int first, int last) {
    int pivot, j, temp, i;
    if (first < last) {
        pivot = first;
        i = first;
        j = last;
        while (i < j) {
            while (Array[i] <= Array[pivot] && i < last)
                i++;
            while (Array[j] > Array[pivot])

```

```

        j--;
        if (i < j) {
            temp = Array[i];
            Array[i] = Array[j];
            Array[j] = temp;
        }
    }
    temp = Array[pivot];
    Array[pivot] = Array[j];
    Array[j] = temp;

    // Use OpenMP tasks for parallel execution
    #pragma omp task
        // creates parallel tasks for sorting
    parallelQuickSort(Array, first, j - 1);

    #pragma omp task
    parallelQuickSort(Array, j + 1, last);

```

//This allows multiple threads to work on different parts of the array concurrently, potentially speeding up the sorting process.

```

    }
}

```

```

void hyperquicksort(int Array[], int node_number) {
    int relative_node, median_key;

    if (node_number == 0) {
        // Host-computer sends each node a distinct subsequence of N/n elements
        #pragma omp parallel for
        for (int i = 0; i < N; i += N / d) {
            quickSort(Array, i, i + N / d - 1);
        }
    }

    // Synchronize before proceeding to ensure sorting is complete
    #pragma omp barrier

    for (int i = d; i >= 1; i--) {
        // Each node sets relative_node according to the least significant i bits of its node number
        relative_node = node_number & ((1 << i) - 1);

        if (relative_node == 0) {

```

```

// Nodes with relative_node 000 find the median key
median_key = Array[N / (1 << i) / 2];

// Each node separates its items into two groups
int left = 0, right = N / (1 << i) - 1;
while (left < right) {
    while (Array[left] <= median_key && left < right)
        left++;
    while (Array[right] > median_key && left < right)
        right--;
    if (left < right) {
        int temp = Array[left];
        Array[left] = Array[right];
        Array[right] = temp;
    }
}

// Synchronize before proceeding to ensure data is correctly partitioned
#pragma omp barrier

// Each node merges the remaining items (parallelized quickSort)
#pragma omp parallel for
for (int j = 0; j < N / (1 << i); j++) {
    quickSort(Array, j * (1 << i), (j + 1) * (1 << i) - 1);
}

// Synchronize before proceeding to ensure merging is complete
#pragma omp barrier
}
}
For sequential quicksort, parallel quicksort, hyperquicksort

```

$N = 10$

```

● saad@DESKTOP-9F70C3D:~/project$ ./main
Unsorted Array: 93 13 73 30 79 31 95 22 26 1
Sequential Quicksort elapsed time: 0.000001 seconds
Sequentially Sorted Array: 1 13 22 26 30 31 73 79 93 95
Parallel Quicksort elapsed time: 0.000263 seconds
Parallaly Sorted Array: 1 13 22 26 30 31 73 79 93 95
Hyper Quicksort elapsed time: 0.000007 seconds
Hyper Quick Sorted Array: 1 13 22 26 30 31 73 79 93 95

```

$N = 100$

```

● saad@DESKTOP-9F70C3D:~/project$ ./main
Sequential Quicksort elapsed time: 0.000009 seconds
Parallel Quicksort elapsed time: 0.000398 seconds
Hyper Quicksort elapsed time: 0.000047 seconds

```

$N = 1000$

```

● saad@DESKTOP-9F70C3D:~/project$ ./main
Sequential Quicksort elapsed time: 0.000123 seconds
Parallel Quicksort elapsed time: 0.002817 seconds
Hyper Quicksort elapsed time: 0.000696 seconds

```

$N = 10,000$

```

● saad@DESKTOP-9F70C3D:~/project$ ./main
Sequential Quicksort elapsed time: 0.002277 seconds
Parallel Quicksort elapsed time: 0.019352 seconds
Hyper Quicksort elapsed time: 0.023395 seconds

```

$N = 100,000$

**2nd Program - Code For Parallel Quick Sort by Regular Sampling:**

This code implements the PSRS (Parallel Sorting by Regular Sampling) algorithm, a parallel sorting algorithm suitable for distributed-memory parallel computers. PSRS consists of the following steps:

1. Partitioning: The input array is partitioned into smaller segments, and each segment is sorted independently using the quicksort algorithm.
2. Sampling: Each processor samples elements from its sorted segment to form a global sample set.
3. Pivoting: Pivots are selected from the global sample set, typically by sorting it and choosing evenly spaced elements.
4. Partitioning (Again): Using the selected pivots, the input array is partitioned into subarrays based on pivot values.
5. Sorting: Each subarray is sorted independently.
6. Merging: The sorted subarrays are concatenated to form the final sorted array.

### Functions :

- The quicksort function is a standard implementation of the quicksort algorithm.
- The PSRS function implements the PSRS algorithm. It partitions the input array into smaller segments, samples elements from each segment, selects pivots, partitions the array based on pivots, sorts subarrays, and finally merges the sorted subarrays.
- The print\_array function is a utility function to print the elements of an array.
- In the main function, different array sizes are tested by initializing arrays with random values, sorting them using the PSRS algorithm, and measuring the time taken for sorting. The array sizes tested range from 10 to 1,000,000, increasing by a factor of 10 each time.

### Code:

```
#include <stdio.h>
#include <stdlib.h>
```

```
void quicksort(int *array, int start, int end) {
    if (start < end) {
        int pivot = array[end];
        int i = start - 1;
        for (int j = start; j < end; j++) {
            if (array[j] <= pivot) {
                i++;
                int temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }
    }
}
```

```

        int temp = array[i + 1];
        array[i + 1] = array[end];
        array[end] = temp;

        quicksort(array, start, i);
        quicksort(array, i + 1, end);
    }
}

```

```

void PSRS(int *array, int n, int p) {
    int size = (n + p - 1) / p;
    int *sample = malloc(sizeof(int) * p * (p - 1));
    int *pivots = malloc(sizeof(int) * (p - 1));
    int *partitions = malloc(sizeof(int) * p);
    int *partition_sizes = malloc(sizeof(int) * p);

    for (int i = 0; i < p; i++) {
        partitions[i] = malloc(sizeof(int) * n); // Allocate max possible size to avoid overflow
        partition_sizes[i] = 0;
    }

    // Sort each segment and sample elements
    for (int i = 0; i < p; i++) {
        int start = i * size;
        int end = (start + size - 1 < n) ? start + size - 1 : n - 1;
        quicksort(array, start, end);
        for (int j = 0; j < p - 1; j++) {
            sample[i * (p - 1) + j] = array[start + (j + 1) * (size / p)];
        }
    }

    // Sort samples and select pivots
    quicksort(sample, 0, p * (p - 1) - 1);
    for (int i = 0; i < p - 1; i++) {
        pivots[i] = sample[(i + 1) * (p - 1) / p];
    }

    // Partition the array using selected pivots
    int current_partition = 0;
    for (int i = 0; i < n; i++) {
        while (current_partition < p - 1 && array[i] > pivots[current_partition]) {
            current_partition++;
        }
    }
}

```

```

    }
    partitions[current_partition][partition_sizes[current_partition]++] = array[i];
}

// Concatenate partitions back into the original array
int index = 0;
for (int i = 0; i < p; i++) {
    quicksort(partitions[i], 0, partition_sizes[i] - 1); // Sort each partition again
    for (int j = 0; j < partition_sizes[i]; j++) {
        array[index++] = partitions[i][j];
    }
    free(partitions[i]); // Free partition memory
}

free(partitions);
free(partition_sizes);
free(sample);
free(pivots);
}

void print_array(const char *label, int *array, int n) {
    printf("%s: ", label);
    for (int i = 0; i < n; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");
}

int main() {

    int p = 4; // Number of partitions

    // Test with different array sizes
    for (int i = 1; i <= 6; i++) {
        int n = 1;
        for (int j = 1; j <= i; j++) {
            n *= 10;

            int *array = malloc(sizeof(int) * n);

            // Initialize array with random values
            for (int j = 0; j < n; j++) {
                array[j] = rand() % 1000;
            }

            clock_t start, end;

```

```

double cpu_time_used;

printf("Array size: %d\n", n);

start = clock();
PSRS(array, n, p);
end = clock();

cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

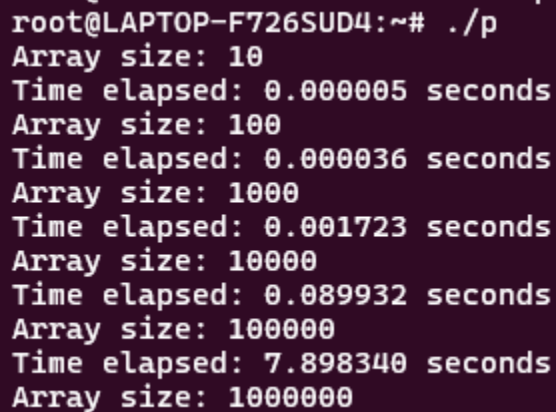
printf("Time elapsed: %.6f seconds\n",

// Free dynamically allocated memory

free(array);
}

return 0;
}

```



```

root@LAPTOP-F726SUD4:~# ./p
Array size: 10
Time elapsed: 0.000005 seconds
Array size: 100
Time elapsed: 0.000036 seconds
Array size: 1000
Time elapsed: 0.001723 seconds
Array size: 10000
Time elapsed: 0.089932 seconds
Array size: 100000
Time elapsed: 7.898340 seconds
Array size: 1000000

```

### 3 program :Quick sort using omp.h

```

#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void swap(int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}

```



```

}

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        #pragma omp task
        quickSort(arr, low, pi - 1);

        #pragma omp task
        quickSort(arr, pi + 1, high);
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int sizes[] = {10, 100, 1000, 10000, 100000, 1000000};
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);

    for (int s = 0; s < num_sizes; s++) {
        int n = sizes[s];
        int *arr = (int *)malloc(n * sizeof(int));
        if (arr == NULL) {
            printf("Memory allocation failed!\n");
            return 1;
        }
    }
}

```

```

    }
    // Initialize array with random values
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 1000;
    }

    printf("Array size: %d\n", n);
    double start_time = omp_get_wtime();

    #pragma omp parallel
    {
        #pragma omp single nowait
        quickSort(arr, 0, n - 1);
    }

    double end_time = omp_get_wtime();

    printf("Time taken: %lf seconds\n", end_time - start_time);

    free(arr);
}

return 0;
}

```

```

root@Beenish:~# nano qs.c
root@Beenish:~# gcc -o qs qs.c -fopenmp
root@Beenish:~# ./qs
Array size: 10
Time taken: 0.032316 seconds
Array size: 100
Time taken: 0.015836 seconds
Array size: 1000
Time taken: 0.036197 seconds
Array size: 10000
Time taken: 0.130615 seconds
Array size: 100000
Time taken: 0.765682 seconds
Array size: 1000000
Time taken: 12.135773 seconds
root@Beenish:~# █

```

## **4th program- Quick sort through Pthread (pos ix)**

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>

// Structure
struct data_set {
    int start_index;
    int end_index;
    int* data;
};

// Function to perform swap operations
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

// Partition function for making partition in array
int partition(int arr[], int left_index, int right_index)
{
    // Choosing pivot element from which we make partition
    int pivot = arr[right_index];
    int i = left_index - 1;

    // Making array as per requirement
    // Arranging elements smaller than pivot on left side and larger than pivot on right side
    for (int j = left_index; j <= right_index - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }

    swap(&arr[i + 1], &arr[right_index]);

    // Returning the partition index
    return i + 1;
}

// Quicksort Function for sorting array
```

```

void* quick_sort(void* data)
{
    struct data_set* info = (struct data_set*)data;

    int left_index = info->start_index;
    int right_index = info->end_index;

    if (left_index < right_index) {
        int index = partition(info->data, left_index, right_index);

        struct data_set* info1 = (struct data_set*)malloc(sizeof(struct data_set));
        struct data_set* info2 = (struct data_set*)malloc(sizeof(struct data_set));

        info1->data = info->data;
        info2->data = info->data;

        info1->start_index = left_index;
        info1->end_index = index - 1;

        pthread_t first_thread;
        pthread_create(&first_thread, NULL, quick_sort, info1);

        info2->start_index = index + 1;
        info2->end_index = right_index;

        pthread_t second_thread;
        pthread_create(&second_thread, NULL, quick_sort, info2);

        pthread_join(first_thread, NULL);
        pthread_join(second_thread, NULL);
    }

    return NULL;
}

int main()
{
    struct timespec start, end;
    double time_taken;

    struct data_set* info = (struct data_set*)malloc(sizeof(struct data_set));

    int sizes[] = {10, 100, 1000, 10000, 100000, 1000000};
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);

```

```

printf("Array Size\tExecution Time (seconds)\n");
for (int i = 0; i < num_sizes; i++) {
    int N = sizes[i];
    int A[N];

    for (int j = 0; j < N; j++) {
        A[j] = rand() % 1000; // Generating random array
    }

    info->data = A;
    info->start_index = 0;
    info->end_index = N - 1;

    clock_gettime(CLOCK_MONOTONIC, &start);
    pthread_t thread_id;
    pthread_create(&thread_id, NULL, quick_sort, info);
    pthread_join(thread_id, NULL);
    clock_gettime(CLOCK_MONOTONIC, &end);

    // Calculate execution time
    time_taken = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1e9;

    printf("%d\t\t%f\n", N, time_taken);
}

return 0;
}

```

```

root@LAPTOP-F726SUD4:~# nano hi.c
root@LAPTOP-F726SUD4:~# nano nao.c
root@LAPTOP-F726SUD4:~# gcc -o nao nao.c
root@LAPTOP-F726SUD4:~# ./nao
Array Size      Execution Time (seconds)
10              0.003846
100             0.018859
1000            0.061247
10000           0.792752
100000          5.266189
1000000         9.750452
root@LAPTOP-F726SUD4:~# 0|

```

## Program 5- quick sort using MPI

```

#include <mpi.h>
#include <stdio.h>

```

```

#include <stdlib.h>
#include <time.h>

void quicksort(int *array, int left, int right) {
    int i = left, j = right;
    int pivot = array[(left + right) / 2];
    int temp;

    while (i <= j) {
        while (array[i] < pivot)
            i++;
        while (array[j] > pivot)
            j--;
        if (i <= j) {
            temp = array[i];
            array[i] = array[j];
            array[j] = temp;
            i++;
            j--;
        }
    }

    if (left < j)
        quicksort(array, left, j);
    if (i < right)
        quicksort(array, i, right);
}

int main(int argc, char *argv[]) {
    int size, rank;
    int n, *array = NULL, *sub_array = NULL;
    int sub_size;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        // Sizes to test
        int sizes[] = {10, 100, 1000, 10000, 100000, 1000000};
        int num_sizes = sizeof(sizes) / sizeof(sizes[0]);

        for (int i = 0; i < num_sizes; i++) {
            n = sizes[i];

```

```

array = (int *)malloc(n * sizeof(int));
for (int j = 0; j < n; j++) {
    array[j] = rand() % 1000000;
}

clock_t start = clock();
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

sub_size = n / size;
sub_array = (int *)malloc(sub_size * sizeof(int));
MPI_Scatter(array, sub_size, MPI_INT, sub_array, sub_size, MPI_INT, 0,
MPI_COMM_WORLD);

quicksort(sub_array, 0, sub_size - 1);

MPI_Gather(sub_array, sub_size, MPI_INT, array, sub_size, MPI_INT, 0,
MPI_COMM_WORLD);

if (rank == 0) {
    quicksort(array, 0, n - 1);
    clock_t end = clock();
    double time_spent = (double)(end - start) / CLOCKS_PER_SEC;
    printf("Array size %d sorted in %f seconds\n", n, time_spent);
    free(array);
}

free(sub_array);
}
} else {
    while (1) {
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n <= 0) break;

        sub_size = n / size;
        sub_array = (int *)malloc(sub_size * sizeof(int));
        MPI_Scatter(NULL, sub_size, MPI_INT, sub_array, sub_size, MPI_INT, 0,
MPI_COMM_WORLD);

        quicksort(sub_array, 0, sub_size - 1);

        MPI_Gather(sub_array, sub_size, MPI_INT, NULL, sub_size, MPI_INT, 0,
MPI_COMM_WORLD);

        free(sub_array);
    }
}

```

```
    }  
}  
  
MPI_Finalize();  
return 0;  
}
```

```
root@Beenish:~# su - mpiuser  
mpiuser@Beenish:~$ nano qmpi.c  
mpiuser@Beenish:~$ mpicc -o qmpi qmpi.c  
mpiuser@Beenish:~$ mpirun -np 4 ./qmpi  
Array size 10 sorted in 0.006458 seconds  
Array size 100 sorted in 0.003776 seconds  
Array size 1000 sorted in 0.001077 seconds  
Array size 10000 sorted in 0.014005 seconds  
Array size 100000 sorted in 0.119503 seconds  
Array size 1000000 sorted in 1.815518 seconds
```



## Project Task 3

### 1. Methodology

#### 1. Quick sort (sequential)

The QuickSort algorithm is a popular and efficient divide-and-conquer sorting technique. The algorithm works by selecting a pivot element from the array and partitioning the remaining elements into two sub-arrays based on their comparison with the pivot. The elements less than the pivot are placed in the left sub-array, while the elements greater than the pivot are placed in the right sub-array. The same process is then recursively applied to the sub-arrays until the base case is reached, i.e., when the sub-array has zero or one element. This divide-and-conquer approach allows the QuickSort algorithm to efficiently sort large datasets with a time complexity of  $O(n \log n)$  on average.

```
function quicksort(array, left, right)
  if left < right
    pivot = partition(array, left, right)
    quicksort(array, left, pivot - 1)
    quicksort(array, pivot + 1, right)

function partition(array, left, right)
  pivot = array[left]
  i = left
  j = right

  while i < j
    while array[i] <= pivot and i < right
      i = i + 1

    while array[j] > pivot
      j = j - 1

    if i < j
      swap array[i] and array[j]

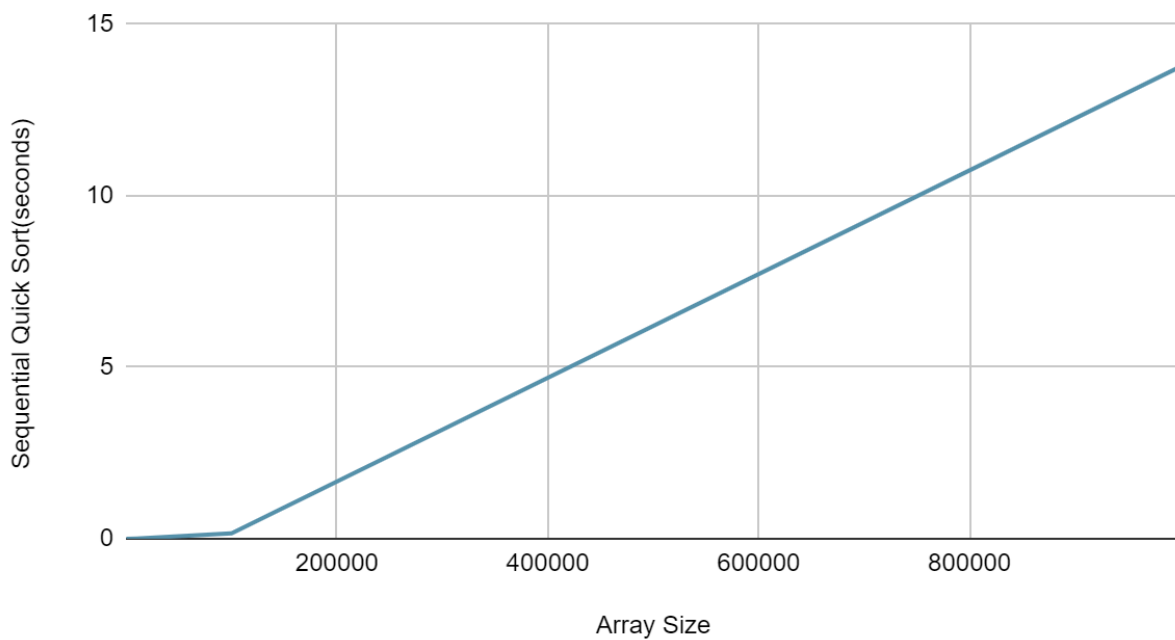
  swap array[left] and array[j]
  return j
```

a high-level overview of the Sequential Quick Sort algorithm

Time taken by Sequential Quick Sort to sort arrays of different sizes:

Array Size	Sequential Quick Sort(seconds)
10	0.000001
100	0.000009
1000	0.000198
10,000	0.002174
100,000	0.155305
1,000,000	13.765057

Sequential Quick Sort(seconds) vs. Array Size



**Analysis:**

The time elapsed for the Sequential Quick Sort algorithm shows that the algorithm performs well for smaller datasets (e.g., 10, 100, and 1000 elements), with execution times ranging from 0.000001 to 0.000198 seconds. However, as the dataset size increases to 10,000, 100,000, and 1,000,000 elements, the execution time also increases significantly, up to 13.765057 seconds for sorting 1,000,000 elements.

- **Parallel processing techniques**

- **Libraries**

To improve the performance of the Quick Sort algorithm for larger datasets, we can leverage parallel processing techniques using libraries such as OpenMP, MPI, and POSIX threads (Pthreads).

**1. *OpenMP*:**

is a widely-used API for shared-memory parallel programming in C, C++, and Fortran. By using OpenMP directives, we can parallelize the recursive calls in the Quick Sort algorithm, allowing multiple threads to sort different sub-arrays simultaneously. This can significantly reduce the overall execution time for large datasets.

```

function quickSort(arr, low, high)
    if low < high
        pi = partition(arr, low, high)

        #pragma omp task
        quickSort(arr, low, pi - 1)

        #pragma omp task
        quickSort(arr, pi + 1, high)

function partition(arr, low, high)
    pivot = arr[high]
    i = low - 1

    for j = low to high - 1
        if arr[j] < pivot
            i++
            swap(arr[i], arr[j])

    swap(arr[i + 1], arr[high])
    return i + 1

function swap(a, b)
    t = a
    a = b
    b = t

function main()
    for each size in array_sizes
        generate input array of size 'size'
        measure and print execution time using OpenMP's 'omp_get_wtime' function

```

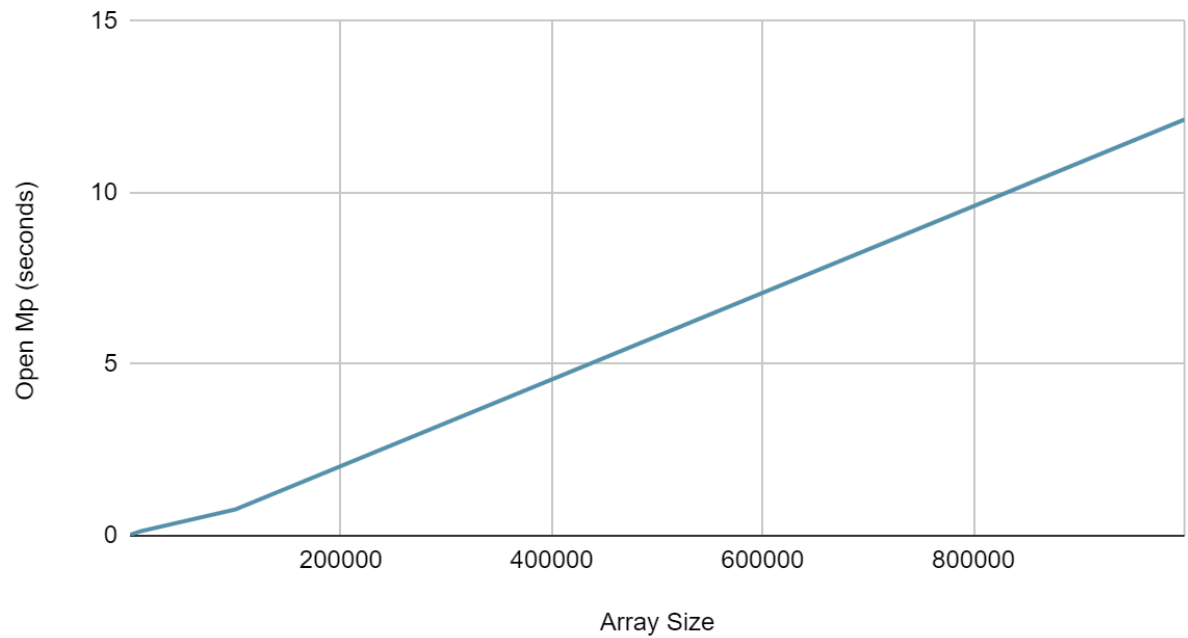
a high-level overview of the open mp algorithm implemented using open Mp

Time taken by Open Mp to sort arrays of different sizes:

Array Size	Open Mp (seconds)
10	0.032316
100	0.015836
1000	0.036197
10,000	0.130615

100,000	0.765682
1,000,000	12.135773

Open Mp (seconds) vs. Array Size



## 2. *POSIX threads (Pthreads):*

Pthreads is a threading library for C and C++ that allows the creation and management of multiple threads within a single process. We can use Pthreads to create multiple threads and assign each thread to sort a specific portion of the input array using the Quick Sort algorithm. The threads can then merge the sorted portions to obtain the final

sorted array.

```
function quick_sort(data)
    left_index = data.start_index
    right_index = data.end_index

    if left_index < right_index
        index = partition(data.data, left_index, right_index)

        info1 = new data_set with start_index = left_index, end_index = index - 1, data =
data.data
        info2 = new data_set with start_index = index + 1, end_index = right_index, data =
data.data

        create thread to execute quick_sort(info1)
        create thread to execute quick_sort(info2)

    join both threads

function partition(array, left_index, right_index)
    pivot = array[right_index]
    i = left_index - 1

    for j = left_index to right_index - 1
        if array[j] < pivot
            i++
            swap array[i] and array[j]

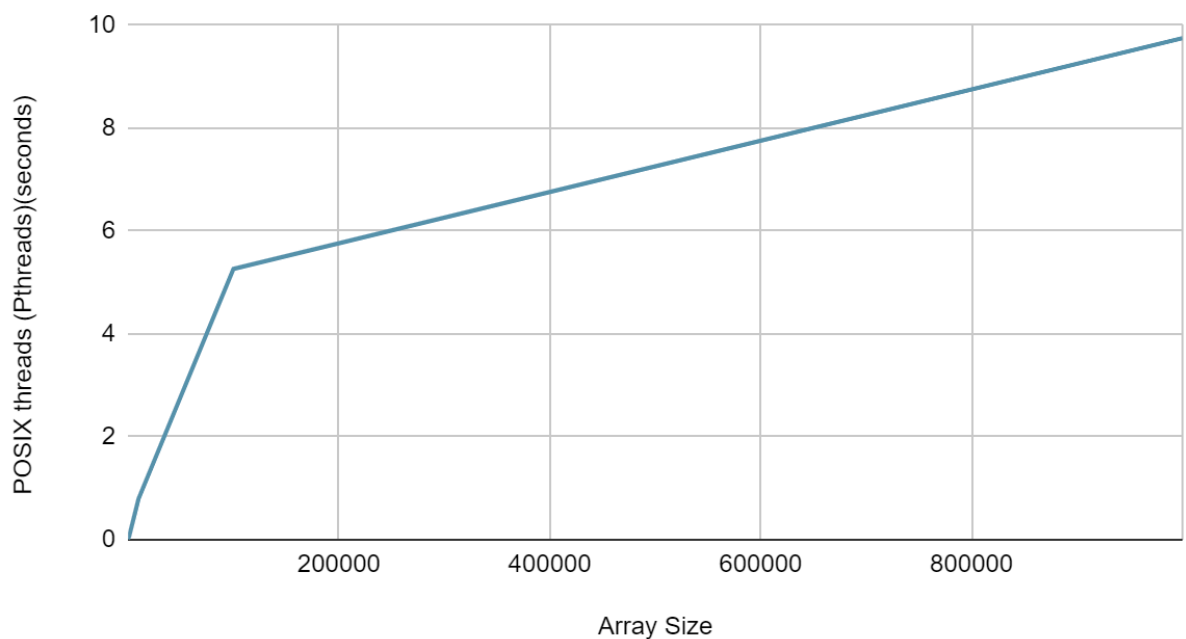
    swap array[i + 1] and array[right_index]
    return i + 1
```

a high-level overview of the parallel Quick Sort algorithm implemented using pthreads

Array Size	POSIX threads (Pthreads)(seconds)
10	0.003846
100	0.018859
1000	0.061247

10,000	0.792752
100,000	5.266189
1,000,000	9.750452

POSIX threads (Pthreads)(seconds) vs. Array Size



### 3. MPI (Message Passing Interface)

is a standardized and portable message-passing system designed for parallel computing on distributed-memory architectures. We can use MPI to implement a master-slave approach for the Quick Sort algorithm, where the master process divides the input array into smaller sub-arrays and distributes them among the slave processes. The slave processes then sort their respective sub-arrays using the Quick Sort algorithm and send the sorted sub-arrays back to the master process, which merges them to obtain the final sorted array.

```

function main()
  if rank == 0
    for each size in array_sizes
      generate input array of size 'size'
      broadcast size to all processes
      scatter input array among all processes
      sort sub-array using quicksort
      gather sorted sub-arrays from all processes
      sort gathered array using quicksort
      measure and print execution time
    else
      while true
        receive size from master process
        if size <= 0, break
        scatter input array from master process
        sort sub-array using quicksort
        gather sorted sub-array to master process

function quicksort(array, left, right)
  pivot = array[(left + right) / 2]
  i = left
  j = right

  while i <= j
    while array[i] < pivot, increment i
    while array[j] > pivot, decrement j
    if i <= j
      swap array[i] and array[j]
      increment i
      decrement j

  if left < j, quicksort(array, left, j)
  if i < right, quicksort(array, i, right)

```

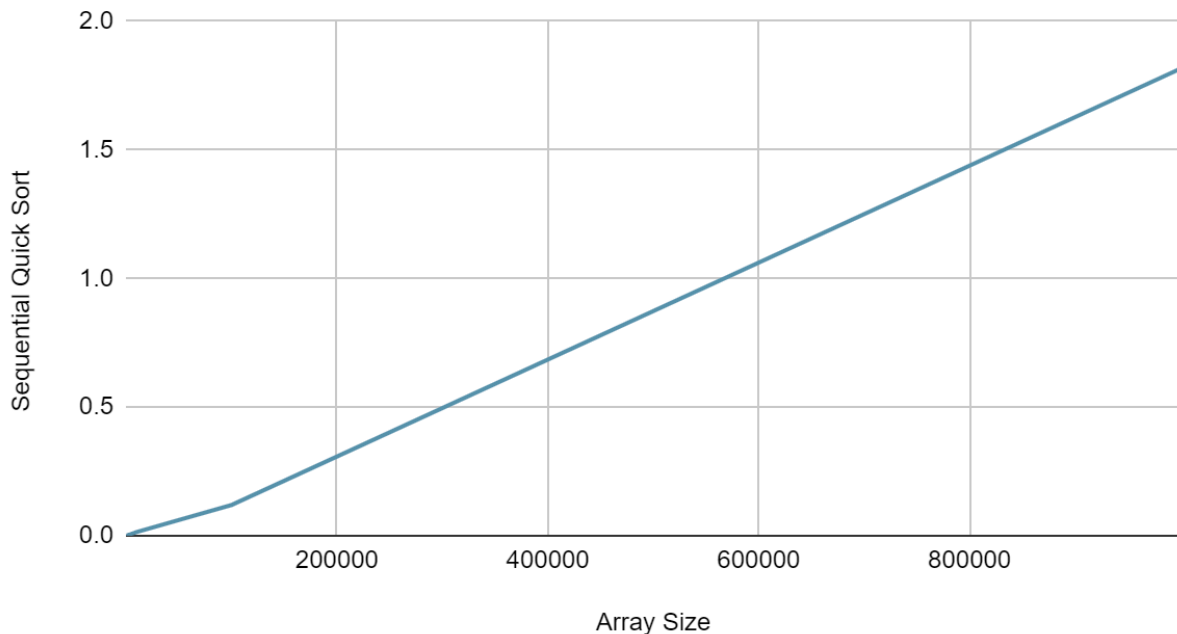
a high-level overview of MPI implemented using MPI

Array Size	Sequential Quick Sort
10	0.006458
100	0.003776
1000	0.001077



10,000	0.014005
100,000	0.119503
1,000,000	1.81551

### Sequential Quick Sort vs. Array Size



#### Conclusion:

We analyzed the performance of the Sequential Quick Sort algorithm and its parallel implementations using OpenMP, Pthreads, and MPI. The Sequential Quick Sort algorithm performed well for smaller datasets but showed increased execution time for larger datasets. To improve efficiency, we explored parallel processing techniques.

OpenMP, Pthreads, and MPI are popular parallel processing libraries that can optimize the Quick Sort algorithm for larger datasets. Each library demonstrated varying improvements in execution time compared to the Sequential Quick Sort algorithm.

In conclusion, leveraging parallel processing techniques can significantly improve the performance of the Quick Sort algorithm for larger datasets. The choice of parallel processing library depends on the specific use case, hardware architecture, and desired level of parallelism. Future work could focus on further optimizing parallel implementations and exploring other parallel processing libraries.

- **Parallel Sorting Algorithms**

- 1. Hyper Quick sort:***

The Hyper Quick Sort algorithm is a parallel sorting algorithm based on the Quicksort method, designed for distributed computing environments. It leverages a P-processor hypercube topology, where each processor receives a portion of the input sequence. The algorithm starts by selecting a pivot and dividing the sequence into two halves, with elements smaller than the pivot on one side and larger on the other. This partitioning process is repeated recursively, with each processor exchanging elements with its neighbors to ensure correct sorting. After several iterations, each processor has a portion of the sorted sequence, which is then locally sorted using a simple Quicksort algorithm. The complexity of Hyper Quick Sort depends on the number of processors ( $P$ ) and the size of the input sequence ( $N$ ), with communication overheads and load balancing considerations affecting its performance. Overall, Hyper Quick Sort offers a parallel approach to sorting large datasets efficiently in distributed computing environments.

```

hyperquicksort(Array[], node_number):
  if node_number == 0:
    // Host-computer sends each node a distinct subsequence of N/n elements
    for i = 0 to N step N/d:
      quickSort(Array, i, i + N/d - 1)

  barrier // Synchronize before proceeding to ensure sorting is complete

  for i = d to 1 step -1:
    relative_node = node_number & ((1 << i) - 1)

    if relative_node == 0:
      // Nodes with relative_node 000 find the median key
      median_key = Array[N / (1 << i) / 2]

      // Each node separates its items into two groups
      left = 0, right = N / (1 << i) - 1
      while left < right:
        while Array[left] <= median_key and left < right:
          left++
        while Array[right] > median_key and left < right:
          right--
        if left < right:
          swap(Array[left], Array[right])

      barrier // Synchronize before proceeding to ensure data is correctly partitioned

      // Each node merges the remaining items (parallelized quickSort)
      for j = 0 to N / (1 << i):
        quickSort(Array, j * (1 << i), (j + 1) * (1 << i) - 1)

      barrier // Synchronize before proceeding to ensure merging is complete

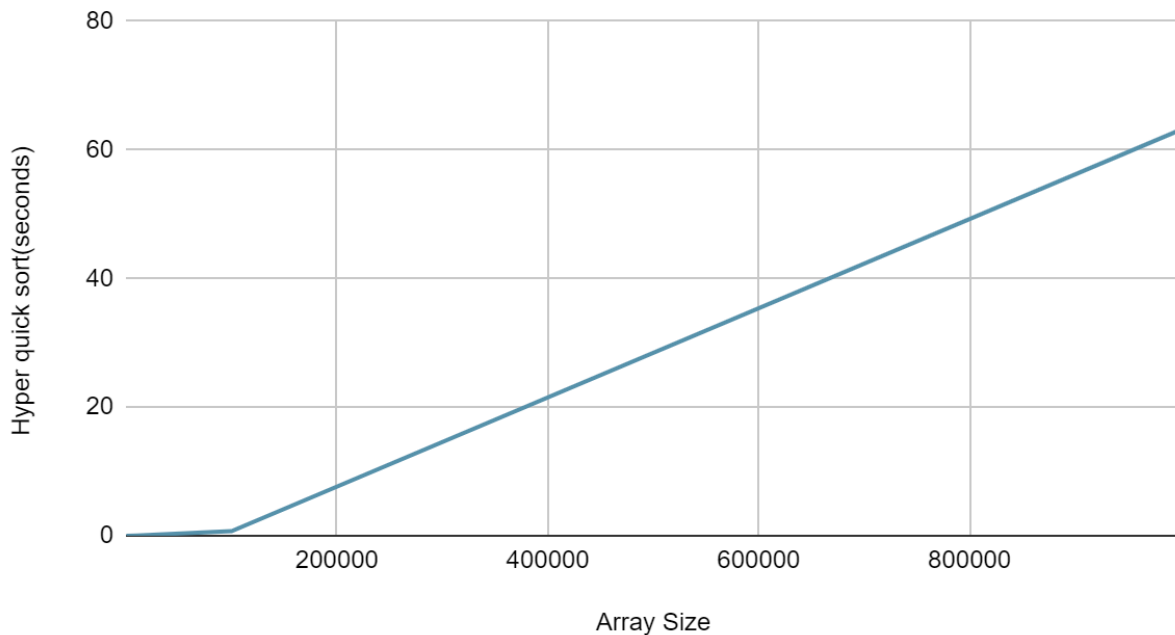
```

Implementation of "hyperquicksort" using OpenMP directives.

Array Size	Hyper quick sort(seconds)
10	0.000241
100	0.000573
1000	0.001325

10,000	0.016115
100,000	0.706276
1,000,000	63.118769

Hyper quick sort(seconds) vs. Array Size



## ***2. Parallel Quick Sort by Regular Sampling***

The Parallel Samplesort (PSRS) algorithm is a parallel sorting algorithm that combines the principles of sample sort and parallel sorting. It is designed to efficiently sort large datasets distributed across multiple processes. In PSRS, each process first sorts its local data using a sequential sorting algorithm like quicksort. Then, each process selects regular samples from its sorted sublist, which are gathered, sorted, and used to determine pivot elements by a single process. These pivot elements are broadcast to the other processes, enabling them to divide their sorted sublists into parts based on the pivot values. An all-to-all communication is performed to migrate the sorted sublist parts to their respective processes. Finally, each process merges its sorted sublists, resulting in a sorted list of the original dataset. PSRS is an efficient and scalable parallel sorting algorithm, particularly for large datasets that cannot fit into the memory of a single process.

```

function PSRS(array, n, p)
    size = (n + p - 1) / p
    allocate memory for sample, pivots, partitions, partition_sizes

    for i in range(0, p)
        initialize partition_sizes[i] to 0

    for i in range(0, p)
        sort each segment of the array and sample elements
        store sampled elements in sample array

    sort sampled elements in sample array
    select pivots from sorted sample array

    partition the array using selected pivots

    concatenate partitions back into the original array
    sort each partition again
    free partition memory

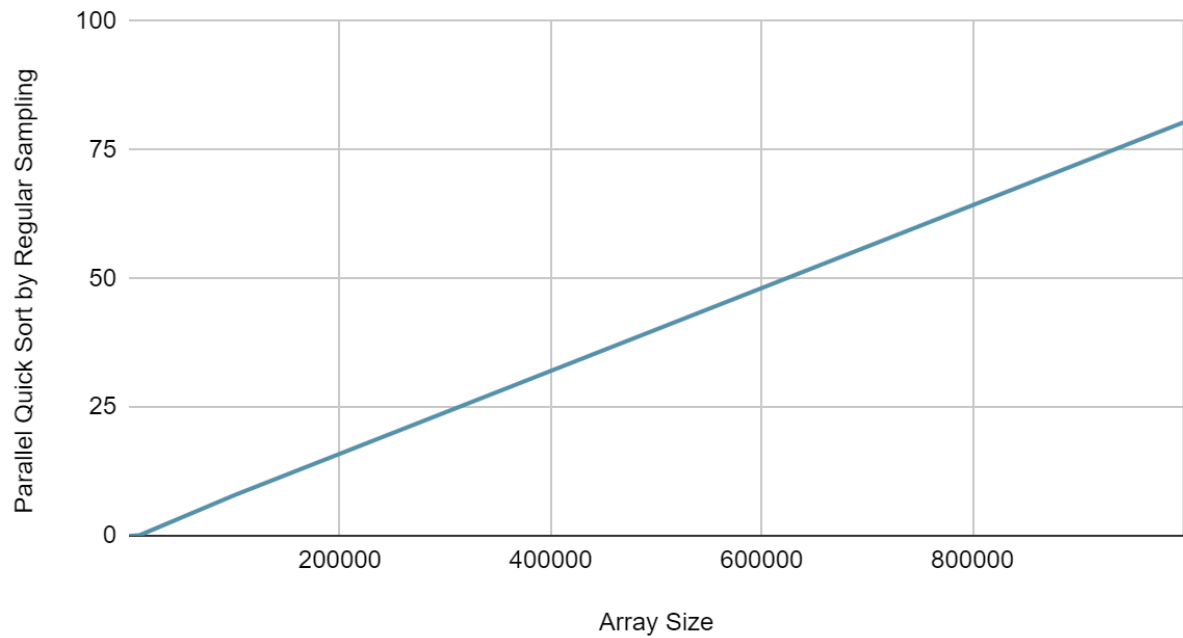
    free allocated memory for sample, pivots, partitions, partition_sizes

```

Pseudocode for psrs

Array Size	Parallel Quick Sort by Regular Sampling
10	0.000005
100	0.000036
1000	0.001723
10,000	0.089932
100,000	7.898340
1,000,000	80.33342

## Parallel Quick Sort by Regular Sampling vs. Array Size



### 2. Flow diagram(s):

- Create one or more diagrams illustrating the flow of data and processes within the code.
- Clearly label each step or component.
- Use shapes and arrows to represent the flow of information.

## 2. Results

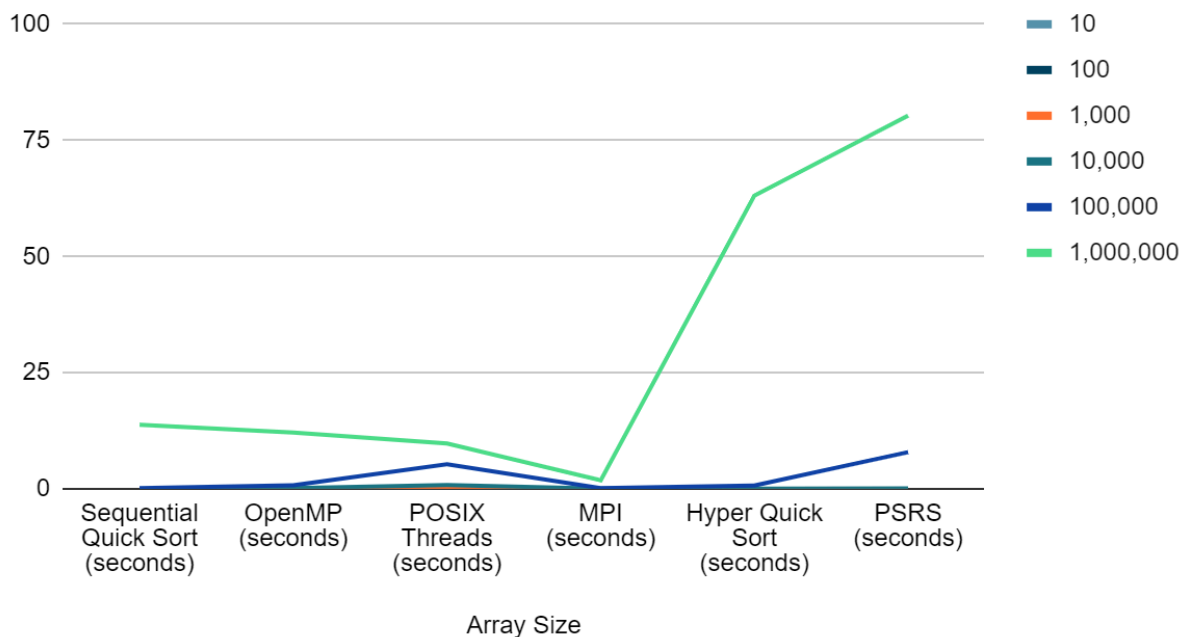
- Tabular representation:

### Combined Table for All Sorting Techniques

Array Size	Sequential Quick Sort (seconds)	OpenMP (seconds)	POSIX Threads (seconds)	MPI (seconds)	Hyper Quick Sort (seconds)	PSRS (seconds)
10	0.000001	0.032316	0.003846	0.006458	0.000241	0.000005
100	0.000009	0.015836	0.018859	0.003776	0.000573	0.000036
1,000	0.000198	0.036197	0.061247	0.001077	0.001325	0.001723

10,000	0.002174	0.130615	0.792752	0.014005	0.016115	0.089932
100,000	0.155305	0.765682	5.266189	0.119503	0.706276	7.898340
1,000,000	13.765057	12.135773	9.750452	1.81551	63.118769	80.33342

10, 100, 1,000, 10,000, 100,000...



## Insights

- Sequential Quick Sort:
  - Performs exceptionally well for small datasets.
  - Execution time increases significantly with larger datasets, making it less efficient for large-scale sorting.
- OpenMP:
  - Shows considerable overhead for very small arrays.
  - Gains efficiency with larger datasets, though it doesn't scale as well as expected for very large datasets (1,000,000 elements).
- POSIX Threads:
  - Offers better performance for medium to large datasets compared to OpenMP.
  - More efficient than Sequential Quick Sort for large arrays but has higher initial overhead for small datasets.
- MPI:

- Shows impressive performance for medium and large datasets.
  - Efficient handling of distributed memory systems.
  - Outperforms Sequential Quick Sort and OpenMP for larger datasets, especially at 1,000,000 elements.
5. Hyper Quick Sort:
    - Very efficient for small to medium datasets.
    - Performance degrades with extremely large datasets due to communication overheads and load balancing issues.
  6. PSRS (Parallel Quick Sort by Regular Sampling):
    - Extremely efficient for small to medium datasets.
    - Performance degrades for very large datasets, similar to Hyper Quick Sort, due to increased communication overhead and complexity in merging sorted sublists.

#### General Insight:

- Parallel sorting techniques such as OpenMP, MPI, Pthreads, Hyper Quick Sort, and PSRS show significant improvements over Sequential Quick Sort for larger datasets.
- MPI offers the most balanced performance across different array sizes, with efficient handling of larger datasets.
- For applications requiring frequent sorting of very large datasets, a distributed approach using MPI or a hybrid parallel approach might be the most effective.
- For smaller datasets, the simplicity and low overhead of Sequential Quick Sort make it the most suitable option.

### 3. Conclusion

To conclude, our report explains the importance of parallelizing algorithms by proving how parallel quicksort algorithms are much better in terms of efficiency compared to sequential quicksort. Various kinds of techniques such as hyper quick sort and random sampling to parallelize quicksort were explained in this report. Along with that time performance of these parallel quick sorts was also analyzed and compared.







