



# **MANIPAL INSTITUTE OF TECHNOLOGY MANIPAL**

*A Constituent Institution of Manipal University*

## **DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING**

### **A Comparative Study of Multithreading and Multiprocessing in Various Searching and Sorting Algorithms**

#### **Project Report**

##### ***Submitted by***

Akshay Gupta - 200905040

Anshita Gera - 200905222

Kaustubh Pandey - 200905142

Rajpreet Lal Das - 200905052

Shubham Srivastava - 200905152

*In partial fulfillment for the award of the degree of*

**Bachelor of Technology**

**In**

**Computer Science and Engineering**

## Department of Computer Science & Engineering



### BONAFIDE CERTIFICATE

Certified that this project report **“A Comparative Study of Multithreading and Multiprocessing in Various Searching and Sorting Algorithms”** is the bonafide work of

Akshay Gupta - 200905040

Anshita Gera - 200905222

Kaustubh Pandey - 200905142

Rajpreet Lal Das - 200905052

Shubham Srivastava - 200905152

who carried out the project under my supervision.

Dr Ashalatha Nayak  
Head of Department  
Department of CSE  
Manipal Institute of Technology  
Manipal, India

Ms Rajashree Krishna  
Professor  
Department of CSE  
Manipal Institute of Technology  
Manipal, India

Submitted to the Viva Voce exam held on

---

Examiner 1

Examiner 2

## **ACKNOWLEDGEMENT**

We would like to thank the Department of Computer Science and Engineering for giving us the opportunity to work on a project to help in our understanding of the coursework.

We would like to thank our teacher Ms. Rajashree Krishna for guiding us through this project and our coursework of Operating Systems. Her teachings have inspired us to take up this implementation. We would also like to express our sincere gratitude to Ms. Vidhya V who also guided us during our labs.

A particular acknowledgement goes to our labmates who helped us by exchanging interesting ideas and information. Finally, we would like to thank our parents for their unwavering support and continuous encouragement in our educational journey.

# TABLE OF CONTENTS

1. Abstract, Keywords and Problem Statement
2. Introduction
  - 2.1 Concepts in Operating Systems
    - 2.1.1 Multithreading
    - 2.1.2 Multiprocessing
  - 2.2 Data Structures and Algorithms
    - 2.2.1 Searching Algorithms
      - 2.2.1.1 Linear Search
      - 2.2.1.2 Binary Search
    - 2.2.2 Sorting Algorithms
      - 2.2.2.1 Bubble Sort
      - 2.2.2.2 Merge Sort
      - 2.2.2.3 Quick Sort
3. Implementation of Multithreading and Multiprocessing in Algorithms
  - 3.1 Linear Search
  - 3.2 Binary Search
  - 3.3 Bubble Sort
  - 3.4 Merge Sort
  - 3.5 Quick Sort
4. Results
5. Conclusion
6. Code Snippets
7. References

## **ABSTRACT**

An algorithm is a detailed, step-by-step process followed in order to accomplish a specific task or to solve a particular problem. The techniques like Multiprocessing and Multithreading both increase a system's computing power and allow us to create efficient strategies to solve real-world problems and applications.

The objective of this project is to perform a comparative study of different sorting and searching techniques under multiprocessed and multithreaded environments. We have selected some widely used sorting and searching techniques such as linear search, binary search, bubble sort, merge sort, and quick sort. The experimental code for its implementation has been done in C language.

## **KEYWORDS**

Multithreading, Multiprocessing, Linear Search, Binary Search, Quick sort, Merge sort, Bubble Sort

## **PROBLEM STATEMENT AND ITS DESCRIPTION**

Appropriate algorithms can have a significant impact on society, and its research can solve real-world problems. Many algorithms have been developed while existing ones have been improved upon all to make them run faster. The efficiency of algorithms can be measured in terms of execution time (complexity) and the amount of memory required. In order to reduce the running time of a given algorithm, techniques like multithreading and multiprocessing are being applied, leading to the evolution of the architecture of computers. It is forecasted that every future innovation will have some advanced functionality. So, we have analyzed different sorting and searching techniques under multiprocessed and multithreaded environments, and compared their performances.

## 2. INTRODUCTION

### 2.1 Concepts in Operating Systems

#### 2.1.1 Multithreading

Multithreading allows the application to divide its task into individual threads. In multi-threads, the same process or task can be done by the number of threads, or we can say that there is more than one thread to perform the task in multithreading. Multitasking is possible with multithreading. Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control.

Real-time applications of multithreading:

1. Gathering information from different web services running in parallel.
  2. Typing MS Word document while listening to music.
  3. Railway ticket reservation system where multiple customers access the server.
  4. Multiple account holders accessing their accounts simultaneously on the server.
- When we insert an ATM card, it starts a thread to perform the operations.

#### 2.1.2 Multiprocessing

Multiprocessing is a mode of operation in which two or more processors in a computer simultaneously process two or more different portions of the same program (set of instructions). Multiprocessing is typically carried out by two or more microprocessors, each of which is in effect a central processing unit (CPU) on a single tiny chip. The primary advantage of a multiprocessor computer is speed, and thus the ability to manage larger amounts of information.

Real-time applications of multiprocessing:

1. Different processors may be used to manage memory storage, data communications, or arithmetic functions.
2. A spin lock is the most basic shared-memory symmetric multiprocessor primitive.
3. It can service various interrupt-driven devices like system timers and I/O devices.

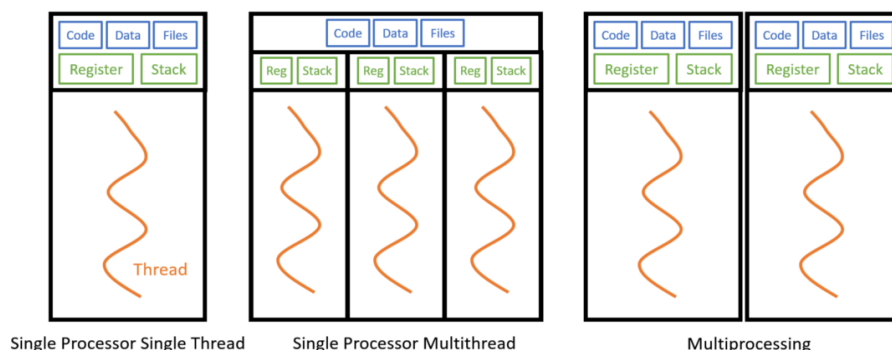


Figure Depicting Multithreading and Multiprocessing

## 2.2 Data Structures and Algorithms

Algorithms play an important role in solving computational problems. In general, an algorithm is a well-defined computational procedure that accepts input and produces output. An algorithm is a sequence of steps or a tool for solving computational problems.

### 2.2.1 Searching Algorithms

Search algorithm, is an efficient algorithm, which performs an important task that locates specific data among a collection of data.

#### 2.2.1.1 Linear Search

Linear search, also known as sequential search, is a method for finding a particular value in a list that consists of checking every one of its elements one at a time and in sequence until the desired one is found. Linear search does not require the collection to be sorted.

#### 2.2.1.2 Binary search

Binary search, also known as half-interval search or logarithmic search, is a search algorithm that finds the position of a target value within a sorted array. It compares the target value to the middle element of the array; if they are unequal, the half in which the target cannot lie is eliminated and the search continues on the remaining half until it is successful.

S. No.	Best Time Complexity	Average Time Complexity	Worst Time Complexity
Linear Search	$O(1)$	$O(n)$	$O(n)$
Binary Search	$O(1)$	$O(\log n)$	$O(\log n)$

where 'n' is the number of elements in the array

Table Depicting the Best-Case, Average-Case, and Worst-Case Time Complexities of Searching Algorithms

## 2.2.2 Sorting Algorithms

### 2.2.2.1 Bubble Sort

Bubble sort is a simple sorting algorithm for a given dataset, it compares the first two elements and if the first one is greater than the second element, it swaps them. This process is performed for each pair of adjacent elements until it reaches the end of the given dataset. The whole process is repeated again until no swaps have occurred on the last pass.

### 2.2.2.2 Merge Sort

It is a recursive algorithm which continuously splits the array in half until it cannot be further divided. This means that if the array becomes empty or has only one element left, the dividing will stop, i.e. it is the base case to stop the recursion. If the array has multiple elements, split the array into halves and recursively invoke the merge sort on each of the halves. Finally, when both halves are sorted, the merge operation is applied. Merge operation is the process of taking two smaller sorted arrays and combining them to eventually make a larger one.

### 2.2.2.3 Quick Sort

This algorithm works by partitioning a given dataset into two parts based on a selected pivot, the low part elements contain values less than the selected pivot while the high part contains values that are higher than the pivot. The algorithm is recursively called with the same concept over every two partitions until no further division is possible after which, the array will be sorted.

S. No.	Best Time Complexity	Average Time Complexity	Worst Time Complexity
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

where 'n' is the number of elements in the array

Table Depicting the Best-Case, Average-Case, and Worst-Case Time Complexities of Sorting Algorithms



## 3. IMPLEMENTATION OF MULTITHREADING AND MULTIPROCESSING IN ALGORITHMS

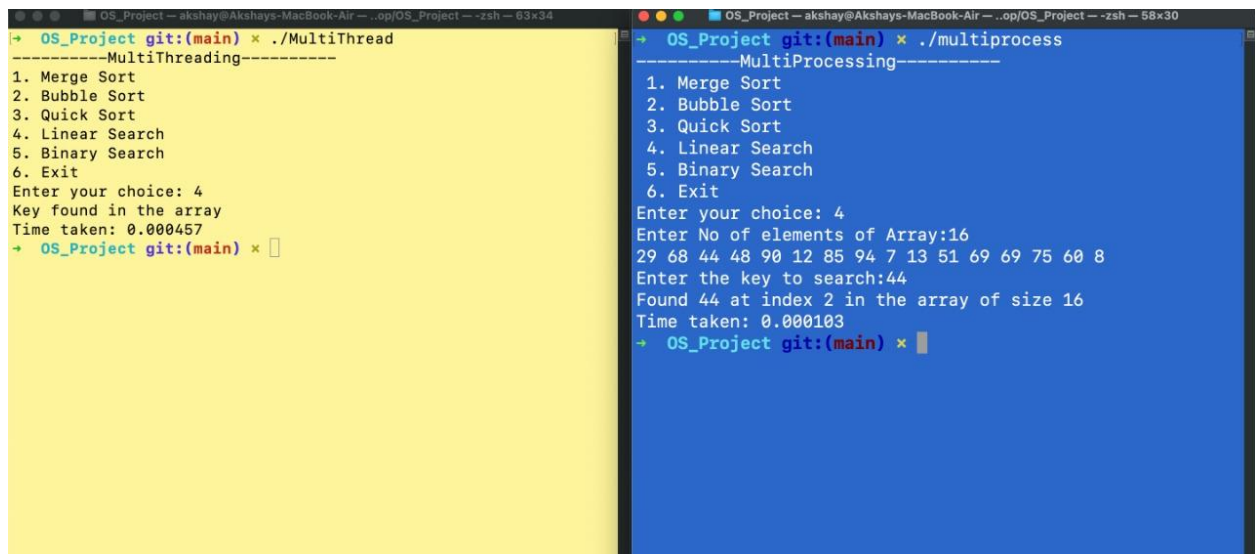
### 3.1 Linear Search

#### Methodology:

The below analysis is for multiprocessing and multithreaded environment:

- For Multithreading, we have used the library of pthread to create two threads, each of which will search for the key value. If found, we set the flag variable to 1 and display the output along with the time taken to search the key element.
- For Multiprocessing, we create two processes which help to find a particular value in a list that consists of checking every one of its elements one at a time and in sequence, until the desired one is found.

#### Output Snippet:



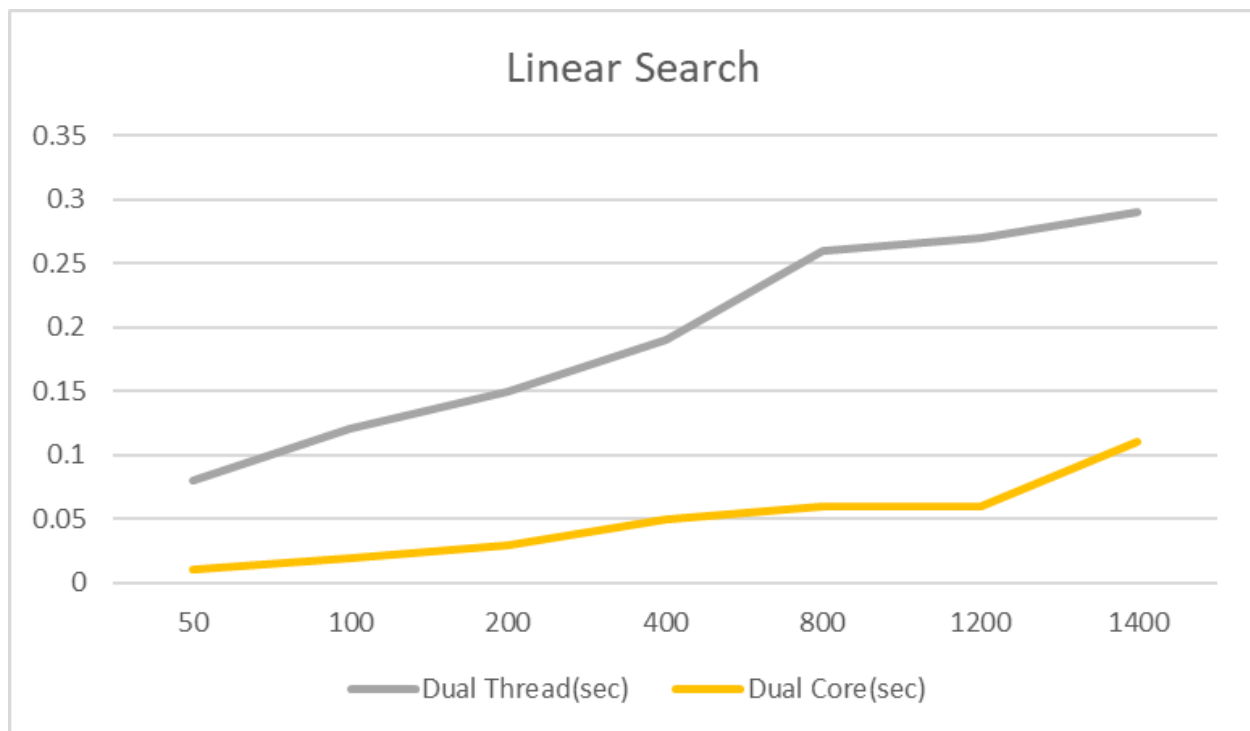
```
OS_Project git:(main) x ./MultiThread
-----MultiThreading-----
1. Merge Sort
2. Bubble Sort
3. Quick Sort
4. Linear Search
5. Binary Search
6. Exit
Enter your choice: 4
Key found in the array
Time taken: 0.000457
OS_Project git:(main) x

OS_Project git:(main) x ./multiprocess
-----MultiProcessing-----
1. Merge Sort
2. Bubble Sort
3. Quick Sort
4. Linear Search
5. Binary Search
6. Exit
Enter your choice: 4
Enter No of elements of Array:16
29 68 44 48 90 12 85 94 7 13 51 69 69 75 60 8
Enter the key to search:44
Found 44 at index 2 in the array of size 16
Time taken: 0.000103
OS_Project git:(main) x
```

The below analysis is for multiprocessing and multithreaded environment:

Array Size	Running Time of Linear Search Dual Thread(ms)	Running Time of Linear Search Dual-core(ms)
50	0.08	0.01
100	0.12	0.02
200	0.15	0.03
400	0.19	0.05
800	0.26	0.06
1200	0.27	0.06
1400	0.29	0.11

The Graphical Demonstration is shown below:



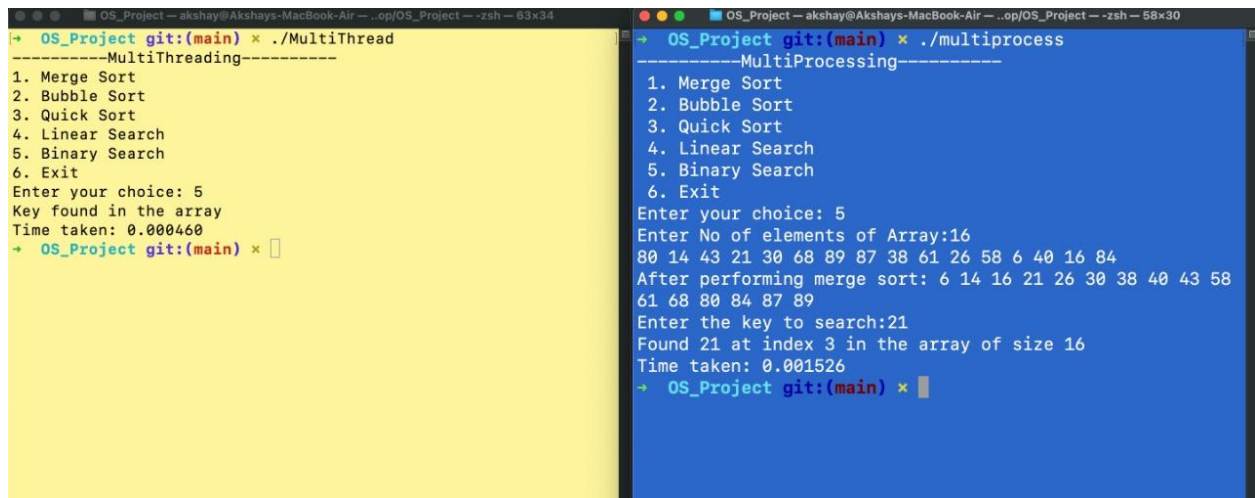
## 3.2 Binary Search

### Methodology:

The below analysis is for multiprocessing and multithreaded environment:

- For Multithreading, we have used the library of pthread to create two threads, each of which will perform binary search individually to search for the key element. If found, we set the found variable to 1 and display the output along with the time taken to search the key element.
- For Multiprocessing, we create two processes which help to find a particular value using binary search. It compares the target value to the middle element of the array, if they are unequal, the half in which the target cannot lie is eliminated and the search continues on the remaining half until it is successful.

### Output Snippet:



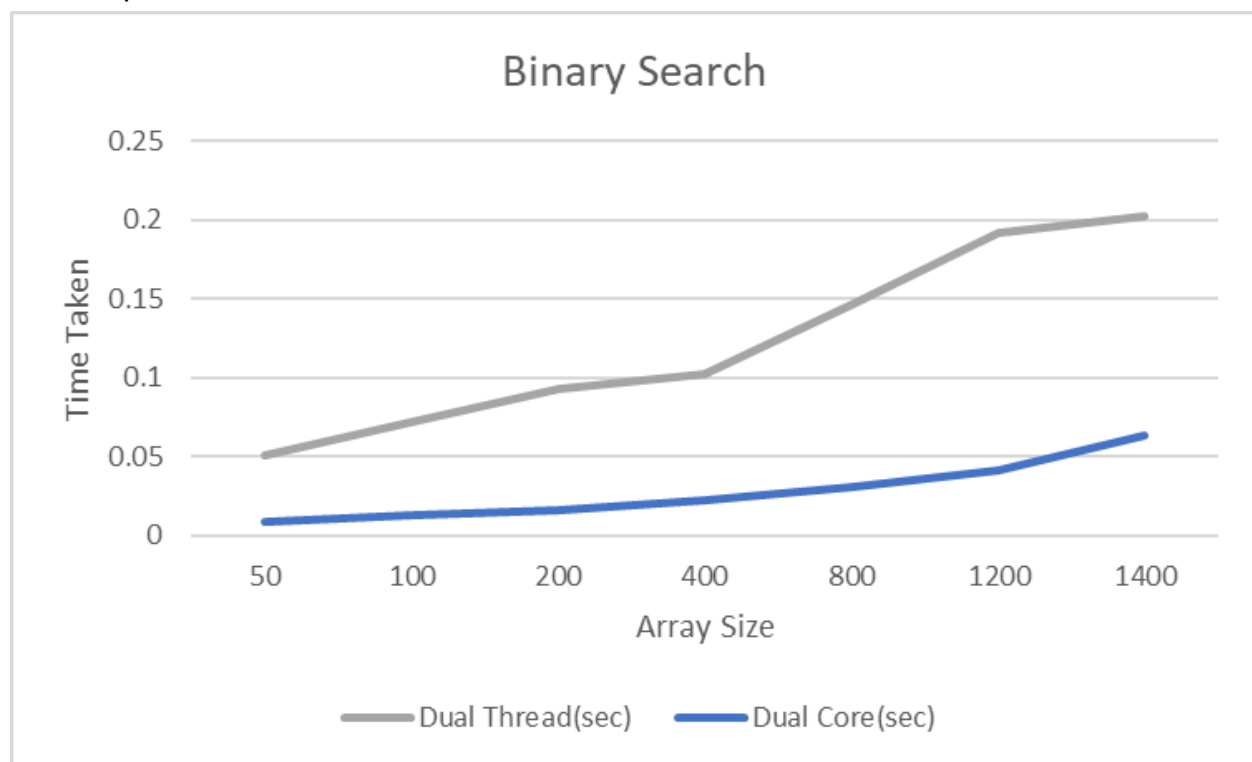
```
OS_Project — akshay@Akshays-MacBook-Air — ..op/OS_Project — -zsh — 63x34
→ OS_Project git:(main) × ./MultiThread
-----MultiThreading-----
1. Merge Sort
2. Bubble Sort
3. Quick Sort
4. Linear Search
5. Binary Search
6. Exit
Enter your choice: 5
Key found in the array
Time taken: 0.000460
→ OS_Project git:(main) ×

OS_Project — akshay@Akshays-MacBook-Air — ..op/OS_Project — -zsh — 58x30
→ OS_Project git:(main) × ./multiprocess
-----MultiProcessing-----
1. Merge Sort
2. Bubble Sort
3. Quick Sort
4. Linear Search
5. Binary Search
6. Exit
Enter your choice: 5
Enter No of elements of Array:16
80 14 43 21 30 68 89 87 38 61 26 58 6 40 16 84
After performing merge sort: 6 14 16 21 26 30 38 40 43 58
61 68 80 84 87 89
Enter the key to search:21
Found 21 at index 3 in the array of size 16
Time taken: 0.001526
→ OS_Project git:(main) ×
```

The below analysis is for multiprocessing and multithreaded environment:

Array Size	Running time of Binary Search Dual Thread(sec)	Running time of Binary Search Dual-core(sec)
50	0.051	0.0090
100	0.072	0.0130
200	0.093	0.0160
400	0.102	0.0220
800	0.147	0.0310
1200	0.192	0.0410
1400	0.203	0.0640

The Graphical Demonstration is shown below:

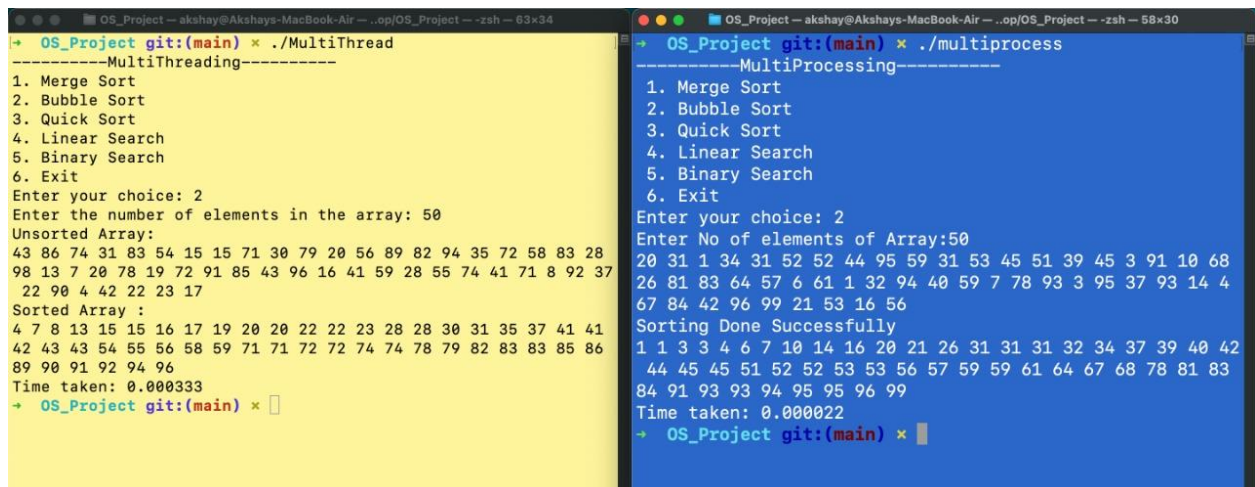


### 3.3 Bubble Sort

#### Methodology:

- For Multithreading, we have used the library of pthread to create two threads, each of which contains the alternative elements from the list. Bubble sort happens in those two threads. The sorted array elements from the threads are then combined.
- For Multiprocessing, we create two processes which help to sort the elements using bubble sort. The sorted array elements from those processes are then combined.

#### Output Snippet:



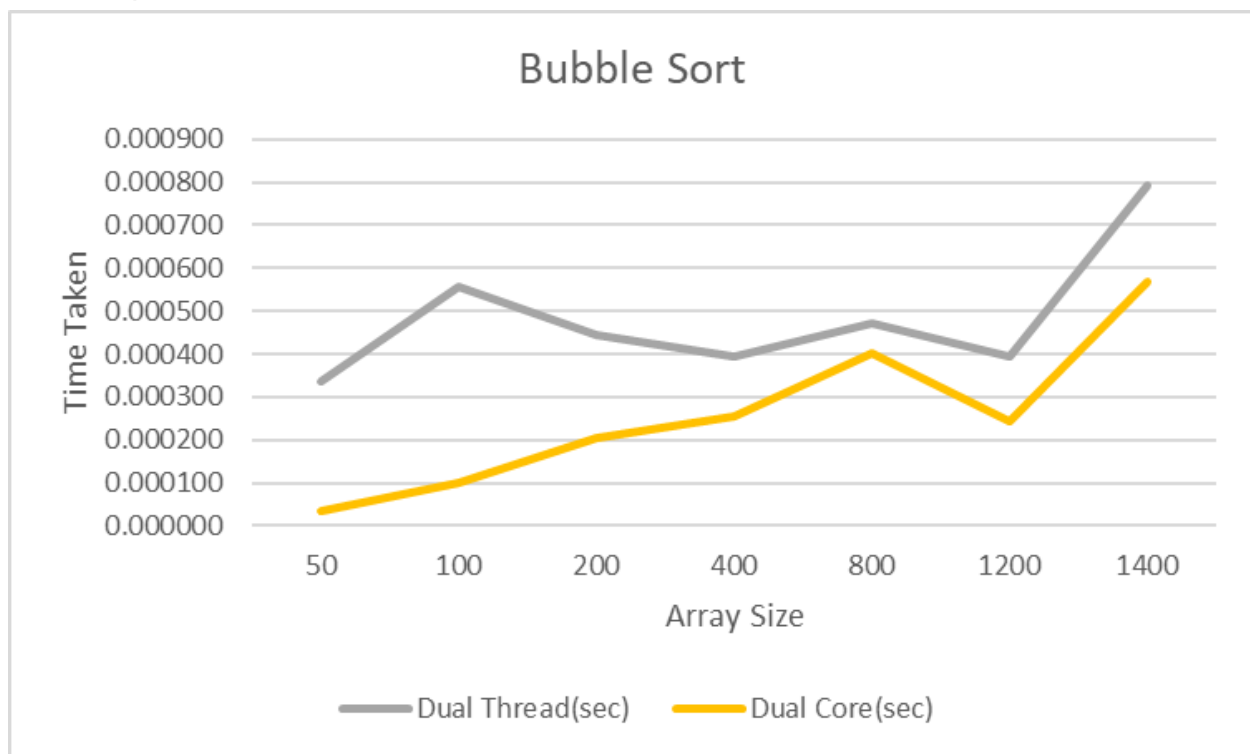
```
OS_Project git:(main) x ./MultiThread
-----MultiThreading-----
1. Merge Sort
2. Bubble Sort
3. Quick Sort
4. Linear Search
5. Binary Search
6. Exit
Enter your choice: 2
Enter the number of elements in the array: 50
Unsorted Array:
43 86 74 31 83 54 15 15 71 30 79 20 56 89 82 94 35 72 58 83 28
98 13 7 20 78 19 72 91 85 43 96 16 41 59 28 55 74 41 71 8 92 37
22 90 4 42 22 23 17
Sorted Array :
4 7 8 13 15 15 16 17 19 20 20 22 22 23 28 28 30 31 35 37 41 41
42 43 43 54 55 56 58 59 71 71 72 72 74 74 78 79 82 83 83 85 86
89 90 91 92 94 96
Time taken: 0.000333
OS_Project git:(main) x

OS_Project git:(main) x ./multiprocess
-----MultiProcessing-----
1. Merge Sort
2. Bubble Sort
3. Quick Sort
4. Linear Search
5. Binary Search
6. Exit
Enter your choice: 2
Enter No of elements of Array:50
20 31 1 34 31 52 52 44 95 59 31 53 45 51 39 45 3 91 10 68
26 81 83 64 57 6 61 1 32 94 40 59 7 78 93 3 95 37 93 14 4
67 84 42 96 99 21 53 16 56
Sorting Done Successfully
1 1 3 3 4 6 7 10 14 16 20 21 26 31 31 31 32 34 37 39 40 42
44 45 45 51 52 52 53 53 56 57 59 59 61 64 67 68 78 81 83
84 91 93 93 94 95 95 96 99
Time taken: 0.000022
OS_Project git:(main) x
```

The below analysis is for multiprocessing and multithreading environment:

Array Size	Running Time of Bubble Sort Dual Thread(sec)	Running Time of Bubble Sort Dual-core(sec)
50	0.000336	0.000034
100	0.000556	0.000102
200	0.000443	0.000203
400	0.000393	0.000256
800	0.000472	0.000401
1200	0.000395	0.000242
1400	0.000792	0.000569

The Graphical Demonstration is shown below:



### 3.4 Merge Sort

#### Methodology:

- For multithreading, we have used the library of pthread to create multiple threads and have joined the threads after merging of split parts while applying merge sort.
- For multiprocessing, we have used the ipc and shm library with fork function to create 2 child processes then we called merge sort for both the processes on equal halves of the number of elements attaching shared memory so that both processes concurrently work on the same memory space. Their sorted results were then merged and the shared memory was detached.

#### Output Snippet:

```
OS_Project git:(main) x ./multithread
-----MultiThreading-----
1. Merge Sort
2. Bubble Sort
3. Quick Sort
4. Linear Search
5. Binary Search
6. Exit
Enter your choice: 1
Enter the size of array: 50
Unsorted Array:
75 75 10 48 2 91 31 47 29 92 80 23 90 95 45 48 29 85 6 75 19 61
47 33 34 5 92 88 40 11 95 45 93 11 52 57 9 51 87 3 47 32 62 98
90 71 2 37 95 49
Sorted Array:
2 2 3 5 6 9 10 11 11 19 23 29 29 31 32 33 34 37 40 45 45 47 47
47 48 48 49 51 52 57 61 62 71 75 75 75 80 85 87 88 90 90 91 92
92 93 95 95 98
Time taken: 0.030144
OS_Project git:(main) x

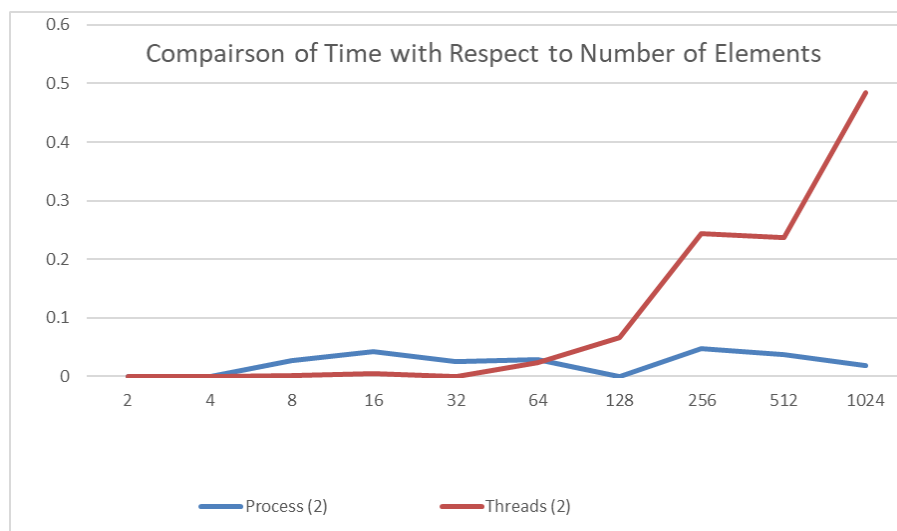
OS_Project git:(main) x ./multiprocess
-----MultiProcessing-----
1. Merge Sort
2. Bubble Sort
3. Quick Sort
4. Linear Search
5. Binary Search
6. Exit
Enter your choice: 1
Enter No of elements of Array:50
39 41 73 36 46 71 25 58 33 57 20 34 53 85 28 88 88 93 46 58 41 4 58 4
1 86 52 5 46 85 39 30 17 59 54 39 74 66 66 51 2 20 79 59 37 12 23 54
7 25 32
Sorting Done Successfully
2 4 5 7 12 17 20 20 23 25 25 28 30 32 33 34 36 37 39 39 41 41 41 4
6 46 46 51 52 53 54 54 57 58 58 58 59 59 66 66 71 73 74 79 85 85 86 8
8 88 93
Time taken: 0.001248
OS_Project git:(main) x
```

The below analysis is for multiprocessing and multithreading environment:

Array Size	Running Time of Merge Sort Dual Thread(sec)	Running Time of Merge Sort Dual-core(sec)
2	0.000604	0.000003
4	0.000753	0.000005
8	0.002328	0.000279

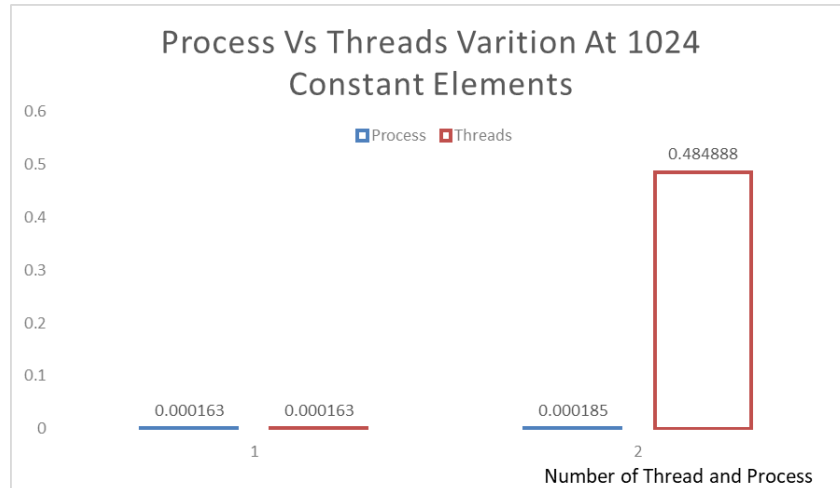
16	0.004585	0.000427
32	0.000385	0.000251
50	0.030144	0.001248
64	0.024131	0.000293
128	0.065857	0.000003
256	0.243865	0.000483
512	0.237705	0.000381
1024	0.484888	0.000185

The Graphical Demonstration is shown below:



The above figure explains a comparison between the process and threads where Number of Threads and Process are both 2 on the sorting of randomly generated elements. The number of elements is increasing in the powers of 2. Here, we have observed that the merge sort with 2-processes is clearly working faster as compared to the merge sort with 2-threads as the number of inputs is increased.





Here, in this figure we have shown a different kind of comparison. Here, the number of elements is kept constant, that is 1024 and the number of threads and processes are increased to observe the change. We can see that when the number of threads and processes were one, no difference was there. But, when the number of processes and threads were increased to two, the number of threads worked more slowly as compared to two processes for the same number of elements. The speed is achieved by dividing the execution time of the sequential version over the execution.

## 3.5 Quick Sort

### Methodology:

- For Multithreading, we have used pthread to make two threads and partitioned the array based on the pivot element. On both parts, we have used quick sort recursively.
- For Multiprocessing, we have created 2 child processes then we called quick sort for both the processes attaching shared memory so that both processes concurrently work on the same memory space. Their sorted results were then combined and the shared memory was detached.

### Output Snippet:

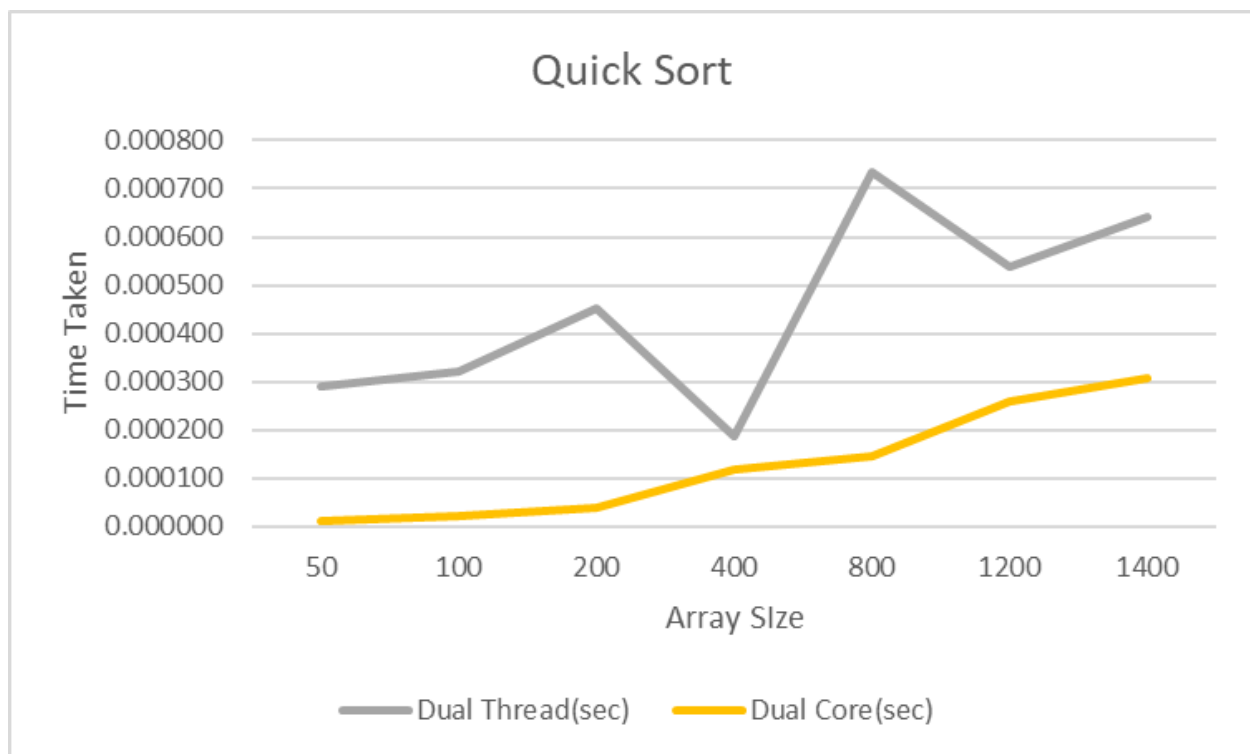
```
OS_Project git:(main) x ./MultiThread
-----MultiThreading-----
1. Merge Sort
2. Bubble Sort
3. Quick Sort
4. Linear Search
5. Binary Search
6. Exit
Enter your choice: 3
Enter the number of elements: 50
Unsorted Array:
75 40 43 94 83 74 71 46 22 77 56 27 87 40 6 42 90 76 35 60 52 9
8 82 2 72 6 14 50 3 54 72 46 88 92 87 61 3 19 20 51 7 86 20 79
47 89 14 89 80 98
Sorted Array :
2 3 3 6 6 7 14 14 19 20 20 22 27 35 40 40 42 43 46 46 47 50 51
52 54 56 60 61 71 72 72 74 75 76 77 79 80 82 83 86 87 87 88 89
89 90 92 94 98 98
Time Taken : 0.0003
OS_Project git:(main) x

OS_Project git:(main) x ./MultiThread
-----MultiThreading-----
1. Merge Sort
2. Bubble Sort
3. Quick Sort
4. Linear Search
5. Binary Search
6. Exit
Enter your choice: 3
Enter the number of elements: 50
Unsorted Array:
10 38 20 90 92 40 91 48 49 34 62 11 59 50 41 16 84 21 41 2
0 73 55 86 99 84 77 62 99 89 41 22 88 1 80 49 32 57 76 80
62 95 11 44 71 84 14 66 9 56 89
Sorted Array :
1 9 10 11 11 14 16 20 20 21 22 32 34 38 40 41 41 41 44 48
49 49 50 55 56 57 59 62 62 62 66 71 73 76 77 80 80 84 84 8
4 86 88 89 89 90 91 92 95 99 99
Time Taken : 0.0002
OS_Project git:(main) x
```

The below analysis is for multiprocessing and multithreading environment:

Array Size	Running Time of Quick Sort Dual Thread(sec)	Running Time of Quick Sort Dual Core(sec)
50	0.000292	0.000013
100	0.000322	0.000021
200	0.000451	0.000039
400	0.000187	0.000118
800	0.000734	0.000147
1200	0.000537	0.000258
1400	0.000643	0.000307

The Graphical Demonstration is shown below:



## 4. RESULTS

- The **linear search** algorithm using multiprocessing code outperforms the multithread code. As the array size grows, so does the number of processors. The p-thread was used for multithread programming, whereas the MPI library was utilised for parallel code.
- The combined **Binary search** with merge sort algorithm with parallel code performs better than the multithread code. As the array size becomes large then the number of processors increases. Multithread code used p-thread, but parallel code was written with MPI library. Moreover, the speedup of the parallel codes on 2, 4, 8, 16, 32, 64, 128, and 143 processors is up to 2.72, and the best is done between 50000 and 500000 dataset sizes, respectively.
- MultiProcessing **Bubble sort** has a faster running time as the number of processors grows. In terms of parallel efficiency, the parallel bubble sort method is more efficient when applied to a small number of processors.
- **Merge sort** is working faster in Multiprocessing as compared to Multithreading. The reason is that multithreading is creating an overhead of synchronization of the threads, secondly as merge sort works on divide and conquer rule so we have to merge all the parts at the end which is time-consuming and there are architecture limitations too.
- The random data in this set is unsuitable for the **Quicksort** method since the initial pivot value divides the data unevenly amongst threads, resulting in a lower improvement in performance compared to standard quicksort. Multiprocessing can only help in a small way. Assuming your machine has a constant number of 'm' processors, the sort will be m times quicker in the ideal scenario. This is due to the fact that the actual sorting effort is now distributed among the cores. Including overhead and unequal job division, it is unlikely to improve significantly.

In conclusion, multithreading will delay our method, but multiprocessing may speed it up slightly, but not significantly enough to make it worthwhile.

## 5. CONCLUSION

In general, simple multithreading will slow down sorting. If we want to retrieve data from the network or multiple disks, then using multiple threads to retrieve it as quickly as possible is a good idea. The actual sorting speed, however, is determined by the number of comparisons and memory operations. A single processor has to do these one at a time, so additional threads simply add overhead from context switching and synchronization.

Multiprocessing also has a significant overhead of stack switching, distinct permissions and separate heaps of processes. Furthermore, since your sort will only have one output, it will need some interprocess communication to join the partially sorted data.

However, multiprocessing can only help by a constant factor. Assuming that the computer has a constant number  $m$  of processors, in the optimal case, the sort will get  $m$  times faster. This is due to the fact that the actual sorting work is now divided across the cores.

To summarize, multithreading will slow down the algorithm, whereas multiprocessing may speed it up.

## Code Snippet: (MultiProcess.c)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <sys/wait.h>
#include "Processes/mergeSortProcess.c"
#include "Processes/linearSearchProcess.c"
#include "Processes/binarySearchProcess.c"
#include "Processes/quickSortProcess.c"
#include "Processes/bubbleSortProcess.c"
void mergeSortRun()
{
    int shmid;
    key_t key = IPC_PRIVATE;
    int *shm_array;
    int length;
    printf("Enter No of elements of Array:");
    scanf("%d", &length);
    // Calculate segment length
    size_t SHM_SIZE = sizeof(int) * length;
    // Create the segment.
    if ((shmid = shmget(key, SHM_SIZE, IPC_CREAT | 0666)) < 0)
    {
        perror("shmget"); _exit(1);
    }
    // Now we attach the segment to our data space.
    if ((shm_array = shmat(shmid, NULL, 0)) == (int *)-1)
    {
        perror("shmat"); _exit(1);
    }
    // Create a random array of given length
    srand(time(NULL));
    TakingInput(shm_array, length);
    printf("\n");
    // startingTime and endingTime for calculating time for merge sort
    clock_t startingTime, endingTime;
    startingTime = clock();
    // Sort the created array
    mergeSort(shm_array, 0, length - 1);
    endingTime = clock();
    // Check if array is sorted or not
    isSorted(shm_array, length);
}
```

```

for (int i = 0; i < length; i++)
{
    printf("%d ", shm_array[i]);
}
printf("\n");
/* Detach from the shared memory now that we are done using it. */
if (shmdt(shm_array) == -1)
{
    perror("shmdt"); _exit(1);
}
/* Delete the shared memory segment. */
if (shmctl(shmid, IPC_RMID, NULL) == -1)
{
    perror("shmctl"); _exit(1);
}
// calculating time taken in applying merge sort
printf("Time taken: %f\n", (endTime - startTime) / (double)CLOCKS_PER_SEC);
exit(0);
}
void bubbleSortRun()
{
    int shmid;
    key_t key = IPC_PRIVATE;
    int *shm_array;
    int length;
    printf("Enter No of elements of Array:");
    scanf("%d", &length);
    // Calculate segment length
    size_t SHM_SIZE = sizeof(int) * length;
    // Create the segment.
    if ((shmid = shmget(key, SHM_SIZE, IPC_CREAT | 0666)) < 0)
    {
        perror("shmget"); _exit(1);
    }
    // Now we attach the segment to our data space.
    if ((shm_array = shmat(shmid, NULL, 0)) == (int *)-1)
    {
        perror("shmat"); _exit(1);
    }
    // Create a random array of given length
    srand(time(NULL));
    TakingInput(shm_array, length);
    printf("\n");
    // startTime and endTime for calculating time for bubble sort
    clock_t startTime, endTime;
    startTime = clock();
    // Sort the created array

```

```

    bubbleSort(shm_array, length);
    endingTime = clock();
    // Check if array is sorted or not
    isSorted(shm_array, length);
    for (int i = 0; i < length; i++)
    {
        printf("%d ", shm_array[i]);
    }
    printf("\n");
    /* Detach from the shared memory now that we are done using it. */
    if (shmdt(shm_array) == -1)
    {
        perror("shmdt"); _exit(1);
    }
    /* Delete the shared memory segment. */
    if (shmctl(shmid, IPC_RMID, NULL) == -1)
    {
        perror("shmctl"); _exit(1);
    }
    // calculating time taken in applying merge sort
    printf("Time taken: %f\n", (endingTime - startingTime) / (double)CLOCKS_PER_SEC); exit(0);
}

void linearSearchRun()
{
    int shmid;
    key_t key = IPC_PRIVATE;
    int *shm_array;
    int length;
    printf("Enter No of elements of Array:");
    scanf("%d", &length);
    // Calculate segment length
    size_t SHM_SIZE = sizeof(int) * length;
    // Create the segment.
    if ((shmid = shmget(key, SHM_SIZE, IPC_CREAT | 0666)) < 0)
    {
        perror("shmget"); _exit(1);
    }
    // Now we attach the segment to our data space.
    if ((shm_array = shmat(shmid, NULL, 0)) == (int *)-1)
    {
        perror("shmat"); _exit(1);
    }
    // Create a random array of given length
    srand(time(NULL));
    TakingInput(shm_array, length);
    printf("\n");
    // startingTime and endingTime for calculating time for merge sort

```



```

    clock_t startingTime, endingTime;
    startingTime = clock();
    int toSearch;
    printf("Enter the key to search:");
    scanf("%d", &toSearch);
    linearSearch(shm_array, length, toSearch);
    endingTime = clock();
    /* Detach from the shared memory now that we are done using it. */
    if (shmdt(shm_array) == -1)
    {
        perror("shmdt"); _exit(1);
    }
    /* Delete the shared memory segment. */
    if (shmctl(shmid, IPC_RMID, NULL) == -1)
    {
        perror("shmctl"); _exit(1);
    }
    // calculating time taken in applying linear search
    printf("Time taken: %f\n", (endingTime - startingTime) / (double)CLOCKS_PER_SEC); exit(0);
}

void quickSortRun()
{
    int shmid;
    key_t key = IPC_PRIVATE;
    int *shm_array;
    int length;
    printf("Enter No of elements of Array:");
    scanf("%d", &length);
    // Calculate segment length
    size_t SHM_SIZE = sizeof(int) * length;
    // Create the segment.
    if ((shmid = shmget(key, SHM_SIZE, IPC_CREAT | 0666)) < 0)
    {
        perror("shmget"); _exit(1);
    }
    // Now we attach the segment to our data space.
    if ((shm_array = shmat(shmid, NULL, 0)) == (int *)-1)
    {
        perror("shmat"); _exit(1);
    }
    // Create a random array of given length
    srand(time(NULL));
    TakingInput(shm_array, length);
    printf("\n");
    // startingTime and endingTime for calculating time for merge sort
    clock_t startingTime, endingTime;
    startingTime = clock();

```

```

// Sort the created array
quickSort(shm_array, 0, length - 1);
endTime = clock();
// Check if array is sorted or not
isSorted(shm_array, length);
for (int i = 0; i < length; i++)
    printf("%d ", shm_array[i]);
printf("\n");
/* Detach from the shared memory now that we are done using it. */
if (shmdt(shm_array) == -1)
{
    perror("shmdt"); _exit(1);
}
/* Delete the shared memory segment. */
if (shmctl(shmid, IPC_RMID, NULL) == -1)
{
    perror("shmctl"); _exit(1);
}
// calculating time taken in applying merge sort
printf("Time taken: %f\n", (endTime - startTime) / (double)CLOCKS_PER_SEC); exit(0);
}

void binarySearchRun()
{
    int shmid;
    key_t key = IPC_PRIVATE;
    int *shm_array;
    int length;
    printf("Enter No of elements of Array:");
    scanf("%d", &length);
    // Calculate segment length
    size_t SHM_SIZE = sizeof(int) * length;
    // Create the segment.
    if ((shmid = shmget(key, SHM_SIZE, IPC_CREAT | 0666)) < 0)
    {
        perror("shmget"); _exit(1);
    }
    // Now we attach the segment to our data space.
    if ((shm_array = shmat(shmid, NULL, 0)) == (int *)-1)
    {
        perror("shmat"); _exit(1);
    }
    // Create a random array of given length
    srand(time(NULL));
    TakingInput(shm_array, length);
    printf("\n");
    // startTime and endTime for calculating time for merge sort
    clock_t startTime, endTime;

```

```

startingTime = clock();
// Sort the created array
mergeSort(shm_array, 0, length - 1);
printf("After performing merge sort: ");
for (int i = 0; i < length; i++)
    printf("%d ", shm_array[i]);
printf("\n");
int toSearch;
printf("Enter the key to search:");
scanf("%d", &toSearch);
binarySearch(shm_array, length, toSearch);
endingTime = clock();
/* Detach from the shared memory now that we are done using it. */
if (shmdt(shm_array) == -1)
{
    perror("shmdt"); _exit(1);
}
/* Delete the shared memory segment. */
if (shmctl(shmid, IPC_RMID, NULL) == -1)
{
    perror("shmctl"); _exit(1);
}
// calculating time taken in applying binary search
printf("Time taken: %f\n", (endingTime - startingTime) / (double)CLOCKS_PER_SEC); exit(0);
}
int main()
{
    int choice;
    do
    {
        printf(" 1. Merge Sort \n 2. Bubble Sort \n 3. Quick Sort \n 4. Linear Search \n 5. Binary Search \n 6. Exit \n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice)
        {
            case 1: mergeSortRun(); break;
            case 2: bubbleSortRun(); break;
            case 3: quickSortRun(); break;
            case 4: linearSearchRun(); break;
            case 5: binarySearchRun(); break;
            case 6: exit(0);
            default: printf("Invalid choice\n");
        }
    } while (choice != 6);
    return 0;
}

```

## Code Snippet: (MultiThread.c)

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <time.h>
#include "Threads/mergeSortThread.c"
#include "Threads/insertionSortThread.c"
#include "Threads/linearSearchThread.c"
#include "Threads/binarySearchThread.c"
#include "Threads/quickSortThread.c"
#include "Threads/bubbleSortThread.c"
void mergeSortRun()
{
    int i, size;
    NODE n;
    // generating random numbers in array
    srand(time(NULL));
    printf("Enter No of elements of Array:\n");
    scanf("%d", &size);
    printf("Unsorted Array:\n");
    for (i = 0; i < size; i++)
    {
        array[i] = rand() % size;
        printf("%d ", array[i]);
    }
    printf("\n");
    n.i = 0;
    n.j = size - 1;
    pthread_t tid;
    int ret;
    // startingTime and endingTime for calculating time for merge sort
    clock_t startingTime, endingTime;
    startingTime = clock();
    ret = pthread_create(&tid, NULL, mergeSort, &n);
    if (ret)
    {
        printf("%d %s - unable to create thread - ret - %d\n", __LINE__, __FUNCTION__, ret); exit(1);
    }
    pthread_join(tid, NULL);
    endingTime = clock();
    printf("Sorted Array:\n");
    for (i = 0; i < size; i++)
        printf("%d ", array[i]);
    printf("\n");
    // calculating time taken in applying merge sort
    printf("Time taken: %f\n", (endingTime - startingTime) / (double)CLOCKS_PER_SEC);
}
```

```

        pthread_exit(NULL);
    }
}

void bubbleSortRun()
{
    int i;
    int N;
    printf("Enter the number of elements in the array: ");
    scanf("%d", &N);
    srand(time(NULL));
    // Filling the array with random integers
    for (i = 0; i < N; i++)
        arr[i] = rand() % 100;
    printf("Unsorted Array:\n");
    for (i = 0; i < N; i++)
        printf("%d ", arr[i]);
    printf("\n");
    pthread_t sorters[2];
    clock_t startingTime, endingTime;
    startingTime = clock();
    printf("Sorted Array :\n");
    for (i = 0; i < 2; i++)
    {
        pthread_create(&sorters[i], NULL, sortArr, (int *)i);
        pthread_join(sorters[i], NULL);
    }
    endingTime = clock();
    printf("Time taken: %f\n", (endingTime - startingTime) / (double)CLOCKS_PER_SEC);
    pthread_exit(NULL);
}

void quickSortRun()
{
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int i;
    srand(time(NULL));
    printf("Unsorted Array:\n");
    for (i = 0; i < n; i++)
    {
        quickarr[i] = rand() % 100;
        printf("%d ", quickarr[i]);
    }
    clock_t t = clock();
    int x = partition(quickarr, 0, n - 1);
    int left_position[2];
    left_position[0] = 0;
    left_position[1] = x - 1;

```

```

    int right_position[2];
    right_position[0] = x + 1;
    right_position[1] = n - 1;
    pthread_t left, right;
    pthread_create(&left, NULL, (void *)quicksort, left_position);
    pthread_create(&right, NULL, (void *)quicksort, right_position);
    pthread_join(left, NULL);
    pthread_join(right, NULL);
    t = clock() - t;
    printf("\nSorted Array : \n");
    for (int i = 0; i < n; i++)
        printf("%d ", quickarr[i]);
    printf("\n");
    printf("Time Taken : %.4f\n", (double)t / (double)CLOCKS_PER_SEC);
    pthread_exit(NULL);
}

void linearSearchRun()
{
    pthread_t thread[THREAD_MAX];
    clock_t startingTime, endingTime;
    startingTime = clock();
    for (int i = 0; i < THREAD_MAX; i++)
        pthread_create(&thread[i], NULL, ThreadSearch, (void *)NULL); // create multiple threads
    for (int i = 0; i < THREAD_MAX; i++)
        pthread_join(thread[i], NULL); // wait until all of the threads are completed
    endingTime = clock();
    if (flag == 1) printf("Key found in the array\n");
    else printf("Key not found\n");
    printf("Time taken: %f\n", (endingTime - startingTime) / (double)CLOCKS_PER_SEC);
    pthread_exit(NULL);
}

void binarySearchRun()
{
    pthread_t threads[MAX_THREAD];
    clock_t startingTime, endingTime;
    startingTime = clock();
    for (int i = 0; i < MAX_THREAD; i++)
        pthread_create(&threads[i], NULL, binary_search, (void *)NULL);
    for (int i = 0; i < MAX_THREAD; i++)
        pthread_join(threads[i], NULL); // wait, to join with the main thread
    endingTime = clock();
    if (found == 1) printf("Key found in the array\n");
    else printf("Key not found\n");
    printf("Time taken: %f\n", (endingTime - startingTime) / (double)CLOCKS_PER_SEC);
    pthread_exit(NULL);
}

```

```

int main()
{
    int choice;
    do
    {
        printf(" 1. Merge Sort\n 2. Bubble Sort\n 3. Quick Sort\n 4. Linear Search\n 5. Binary Search\n 6. Exit \n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice)
        {
            case 1: mergeSortRun();break;
            case 2: bubbleSortRun();break;
            case 3: quickSortRun();break;
            case 4:linearSearchRun(); break;
            case 5: binarySearchRun();break;
            case 6: exit(0);
            default: printf("Invalid choice\n"); break;
        }
    } while (choice != 6);
    return 0;
}

```

## REFERENCES

1. Operating System Concepts Ninth Edition by Abraham Silberschatz, Peter Baer Galvin, Greg Gagne
2. <https://androidmaniacom.wordpress.com/2016/12/16/multithreading-realtime-examples/#:~:text=Games%20are%20very%20good%20examples,multiple%20customers%20accessing%20the%20server>
3. [https://www.researchgate.net/publication/332105787\\_Effects\\_of\\_Multi-core\\_Processors\\_on\\_Linear\\_and\\_Binary\\_Search\\_Algorithms?enrichId=rgreq-61503a64725e148a2de44df9b1e43232-XXX&enrichSource=Y292ZXJQYWdlOzMzMjEwNTc4NztBUzoxMDI2MzM4NjcyNjI3NzE0QDE2MjE3MDk2MTY1MzY%3D&el=1\\_x\\_2&\\_esc=publicationCoverPdf](https://www.researchgate.net/publication/332105787_Effects_of_Multi-core_Processors_on_Linear_and_Binary_Search_Algorithms?enrichId=rgreq-61503a64725e148a2de44df9b1e43232-XXX&enrichSource=Y292ZXJQYWdlOzMzMjEwNTc4NztBUzoxMDI2MzM4NjcyNjI3NzE0QDE2MjE3MDk2MTY1MzY%3D&el=1_x_2&_esc=publicationCoverPdf)
4. <https://pruthvishetty.com/wp-content/uploads/2017/03/Algorithms-Final-Paper.pdf>
5. [https://www.researchgate.net/publication/334131761\\_Performance\\_Evaluation\\_of\\_Parallel\\_Bubble\\_Sort\\_Algorithm\\_on\\_Supercomputer\\_IMAN1?enrichId=rgreq-713c3a3de3f05dd466e8c39166d5974f-XXX&enrichSource=Y292ZXJQYWdlOzMzNDZlMTc2MTtBUzo3NzU3OTE2OTY2ODMwMTJAMTU2MTk3NDU2Mjg3MQ%3D%3D&el=1\\_x\\_2&\\_esc=publicationCoverPdf](https://www.researchgate.net/publication/334131761_Performance_Evaluation_of_Parallel_Bubble_Sort_Algorithm_on_Supercomputer_IMAN1?enrichId=rgreq-713c3a3de3f05dd466e8c39166d5974f-XXX&enrichSource=Y292ZXJQYWdlOzMzNDZlMTc2MTtBUzo3NzU3OTE2OTY2ODMwMTJAMTU2MTk3NDU2Mjg3MQ%3D%3D&el=1_x_2&_esc=publicationCoverPdf)