The Faculty of Information and Communication Technology

Mahidol University

ITCS443 Parallel and Distributed Systems

Semester 1, 2023

Group-Project:

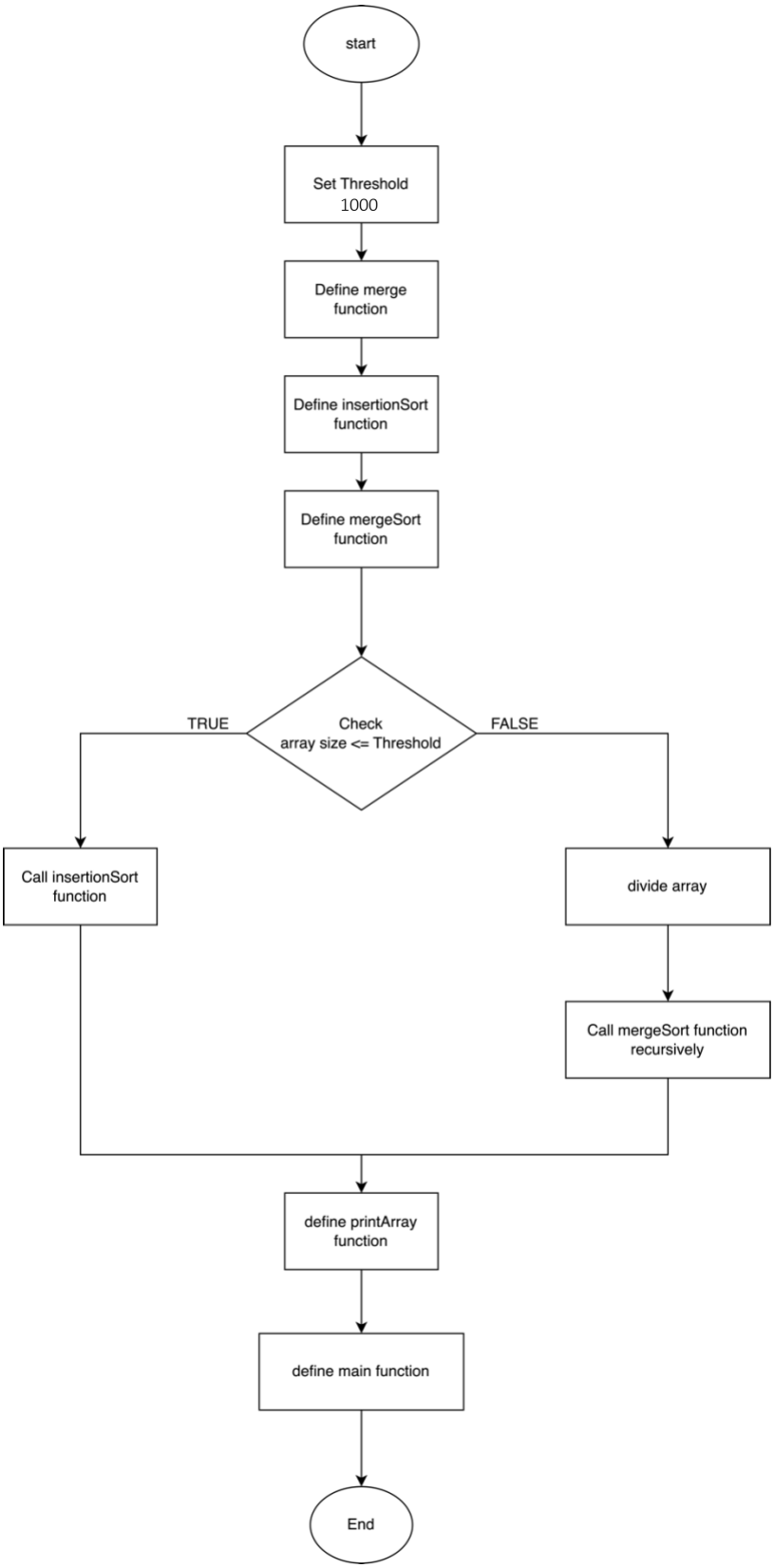
Parallel Merge Sort via OpenMP

Submission Date:	October 21, 2023
Section:	1
Student ID / Name:	6488179 Ponnapassorn Iamborisut
	6488181 Thadeeya Duangkaew
	6488210 Ravikarn Jarungjitvittawas
Instructors:	Dr. Sudsanguan Ngamsuriyaroj
	(sudsanguan.nga@mahidol.ac.th)

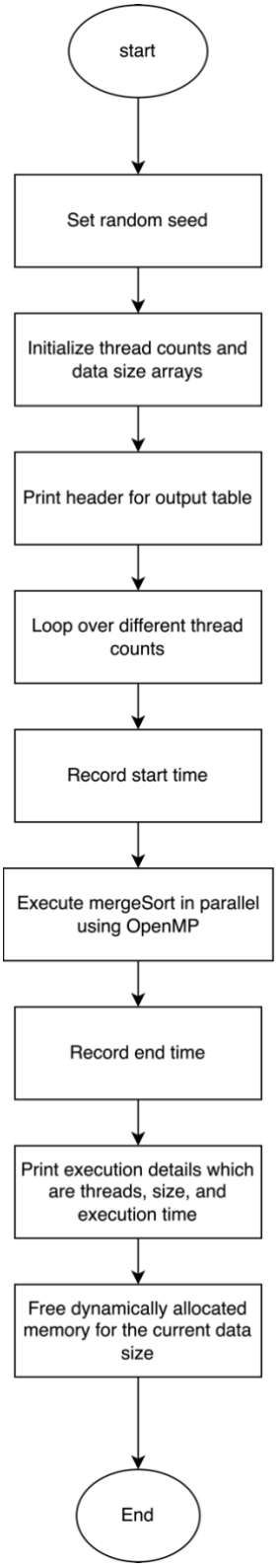
Table of Contents

Explanation the algorithm and implementation

Flowchart:



In main function:



Data Structures that our group use for Merge Sort Algorithm:

- **Array**: The array to be sorted.
- **Thread count**: The number of threads to use for parallel execution.
- **Threshold**: The minimum size of a sub-array for which insertion sort will be used instead of merge sort.

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h> // OpenMP library for parallel programming
#include <time.h>

#define THRESHOLD 1000 // Threshold for using local sorting

// Function to merge two sub-arrays of arr[]
void merge(int arr[], int left, int mid, int right)
{
    //n1 and n2 is to calculate the sizes of two sub-arrays to be merged.
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int *leftArr = (int *)malloc(n1 * sizeof(int));
    int *rightArr = (int *)malloc(n2 * sizeof(int));

    int i;
    for (i = 0; i < n1; i++)
        leftArr[i] = arr[left + i];

    int j;
    for (j = 0; j < n2; j++)
        rightArr[j] = arr[mid + 1 + j];

    //Merging the Two Sub-arrays
    i = 0;
    j = 0;
    int k;
    k = left;
    while (i < n1 && j < n2)
    {
```

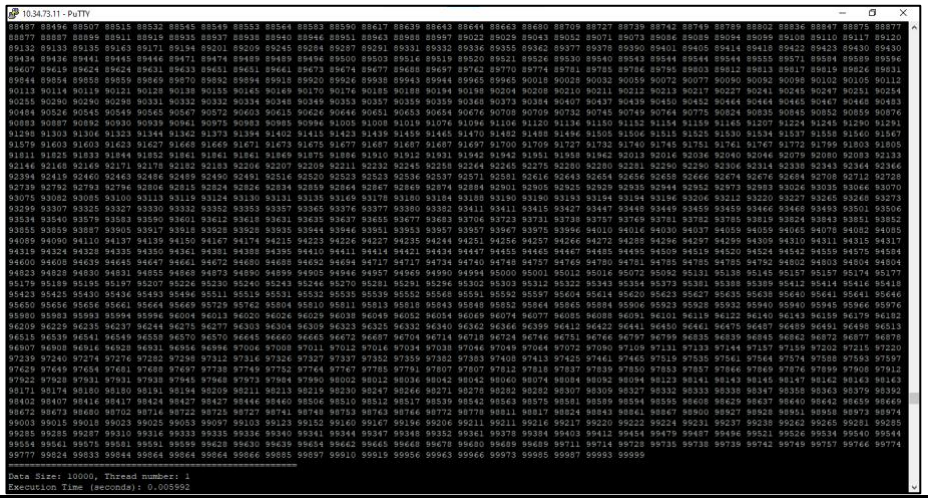
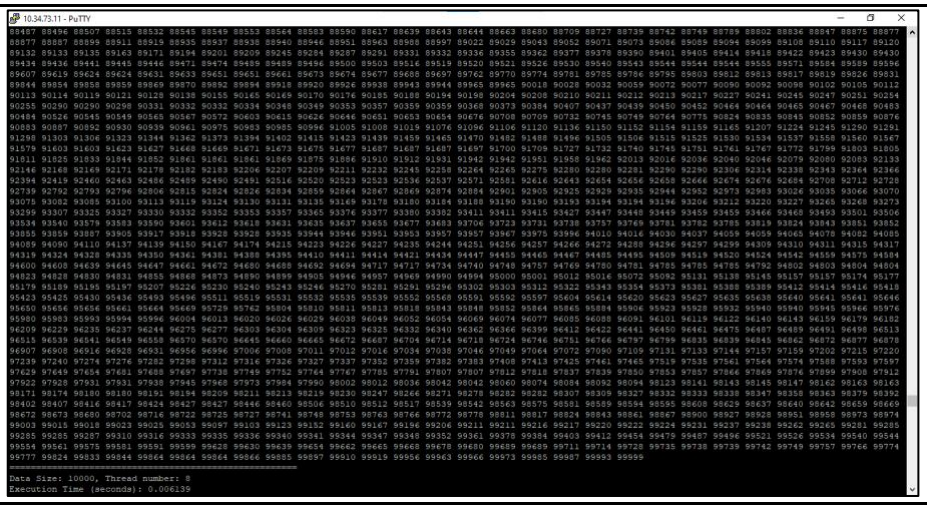
```
        if (leftArr[i] <= rightArr[j])
        {
            arr[k] = leftArr[i];
            i++;
        }
        else
        {
            arr[k] = rightArr[j];
            j++;
        }
        k++;
    }
    while (i < n1)

    {
        arr[k] = leftArr[i];
        i++;
        k++;
    }
    while (j < n2)
    {
        arr[k] = rightArr[j];
        j++;
        k++;
    }
    // Free dynamically allocated memory
    free(leftArr);
    free(rightArr);
}

// Function to perform insertion sort on a sub-array of arr[]
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        // sorting by using insertion sort algorithm
        key = arr[i];
        j = i - 1;
```

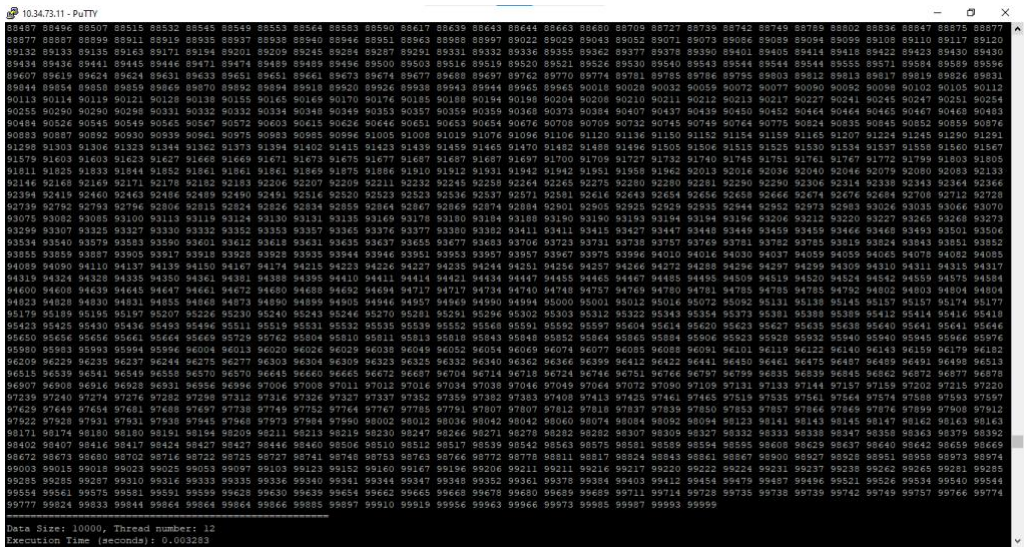
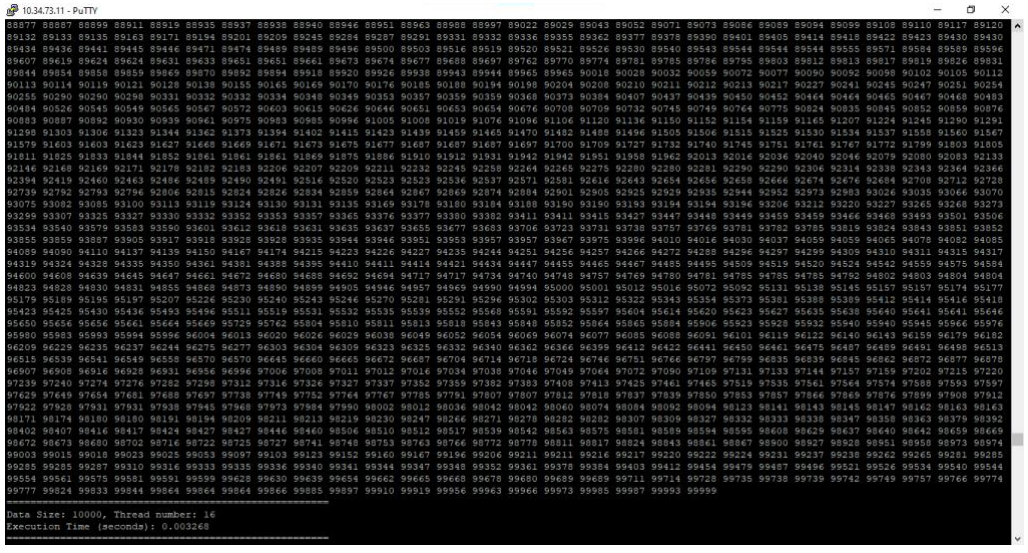
```
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
// Function to perform merge sort on a sub-array of arr[]
void mergeSort(int arr[], int left, int right)
{
    if (left < right)
    {
        //mid index is calculated as the average of left and right, using
integer arithmetic to prevent overflow.
        int mid = left + (right - left) / 2;

        //THRESHOLD Check for Insertion Sort:
        if (right - left + 1 <= THRESHOLD)
        {
            insertionSort(arr + left, right - left + 1);
        }
        else
        {
            #pragma omp task //executed in parallel by the OpenMP runtime.
            mergeSort(arr, left, mid);

            #pragma omp task
            mergeSort(arr, mid + 1, right);

            #pragma omp taskwait //used to wait for all the parallel tasks
to complete before proceeding to the next step.
            merge(arr, left, mid, right);
        }
    }
}

// Function to print an array
void printArray(int arr[], int size)
```

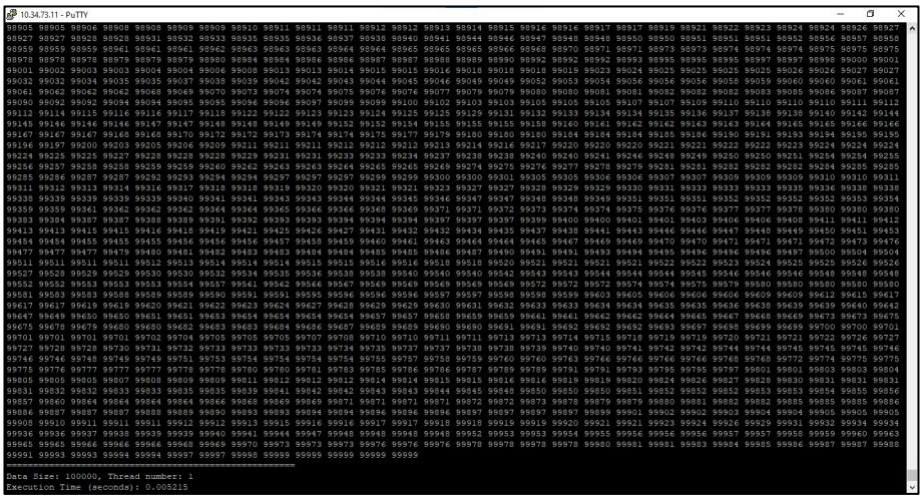
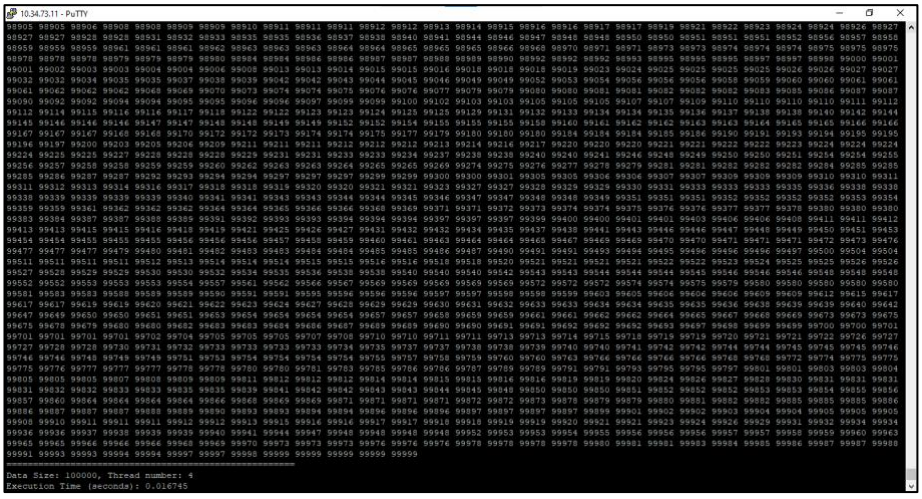
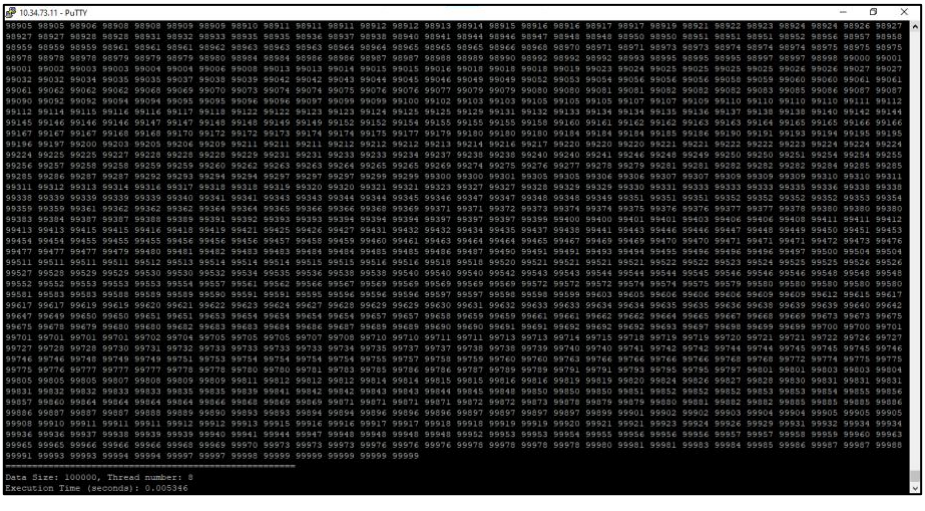
```
{
    int i;
    printf("Sorted Data: ");
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main()
{
    srand(123456);  // Set random seed
    int num_threads[] = {1, 4, 8, 12, 16};  // Array for thread counts
    int data_sizes[] = {10000, 100000, 500000, 1000000, 2000000}; // Array
for data sizes
    printf("Data Size (N) | Num Threads | Execution Time (seconds)\n");
    printf("=============================================================\n");
    int i, t;
    for (i = 0; i < 5; i++) // Loop over different data sizes
    {
        int size = data_sizes[i];                    // Current data size
        int *arr = (int *)malloc(size * sizeof(int)); // Array to be
sorted
        // Generate random numbers for the array
        int j;
        for (j = 0; j < size; j++)
        {
            arr[j] = rand() % 100000; // Set each element randomly to 0-
99999
        }
        for (t = 0; t < 5; t++) // Loop over different thread counts
        {
            double start_time = omp_get_wtime(); // Record start time
            #pragma omp parallel num_threads(num_threads[t]) // Parallel
execution with specified thread count
            {
                #pragma omp single                          // Single
directive to start parallel execution with one thread
                mergeSort(arr, 0, size - 1); // Perform merge sort in
parallel
```

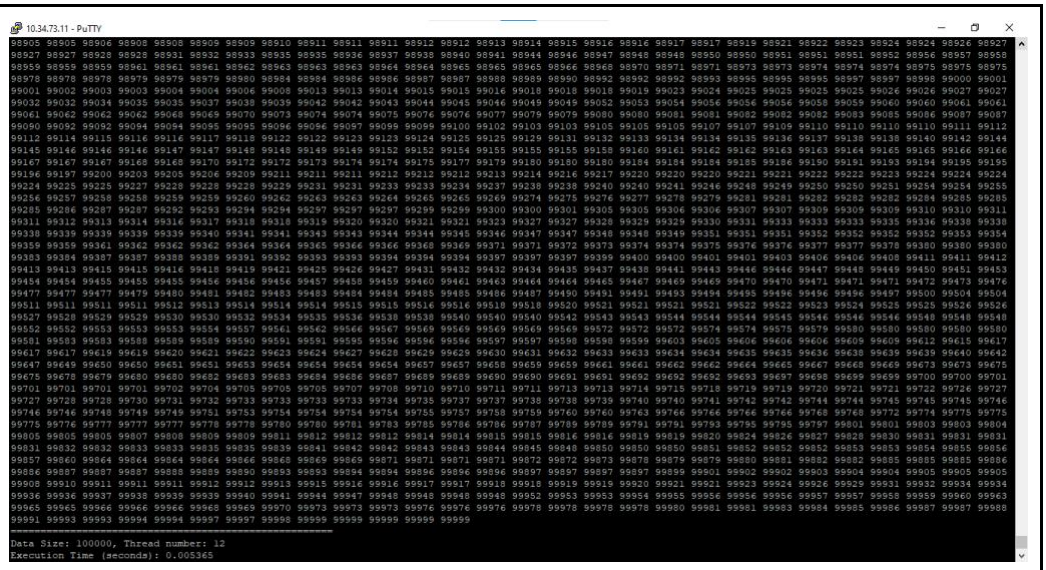
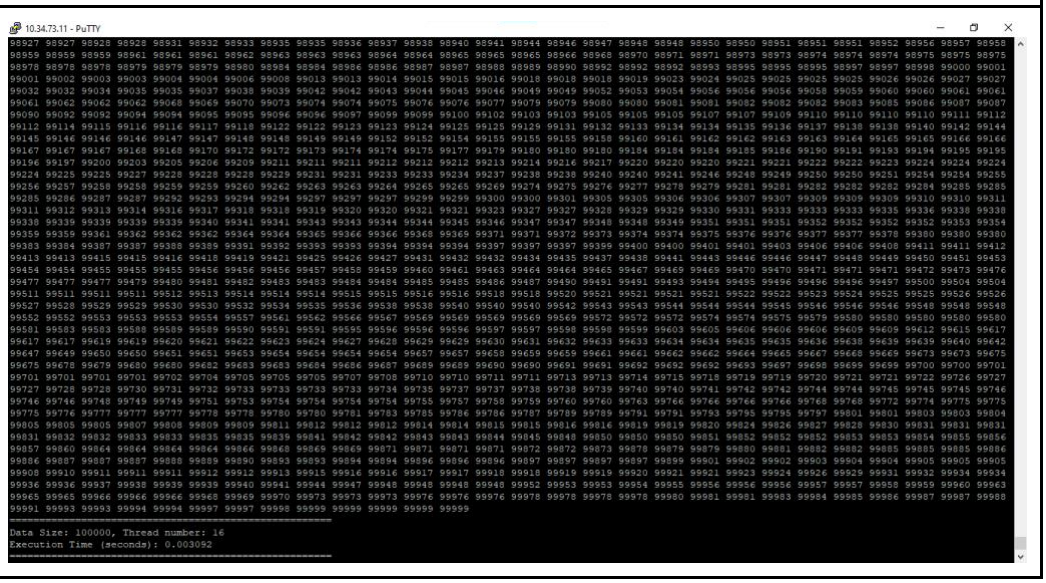
```
            }

            double end_time = omp_get_wtime(); // Record end time

            // Uncomment printArray function to print data that is already
sorted by merge sort.
            //printArray(arr, size);

//printf("=======================================================\n");
            //printf("Data Size: %d, Thread number: %d\n", size,
num_threads[t]);
            //printf("Execution Time (seconds): %lf\n", end_time -
start_time);

            // Print execution details (threads, size, and execution time)

            printf("%10d    | %6d      |     %lf\n", size, num_threads[t],
end_time - start_time);

            if (num_threads[t] == 16)
            {

printf("=======================================================\n");
            }
        }
        // Free dynamically allocated memory for this data size
        free(arr);
    }

    return 0;
}
```

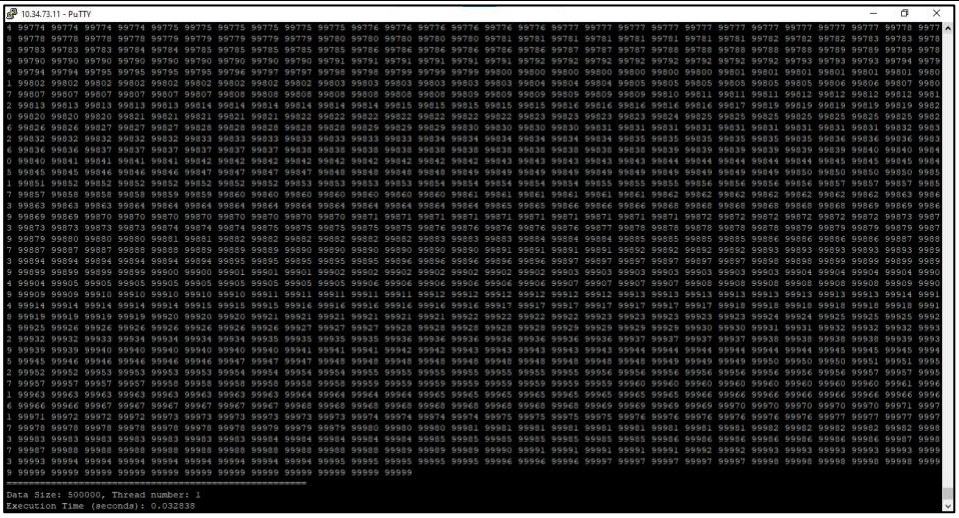
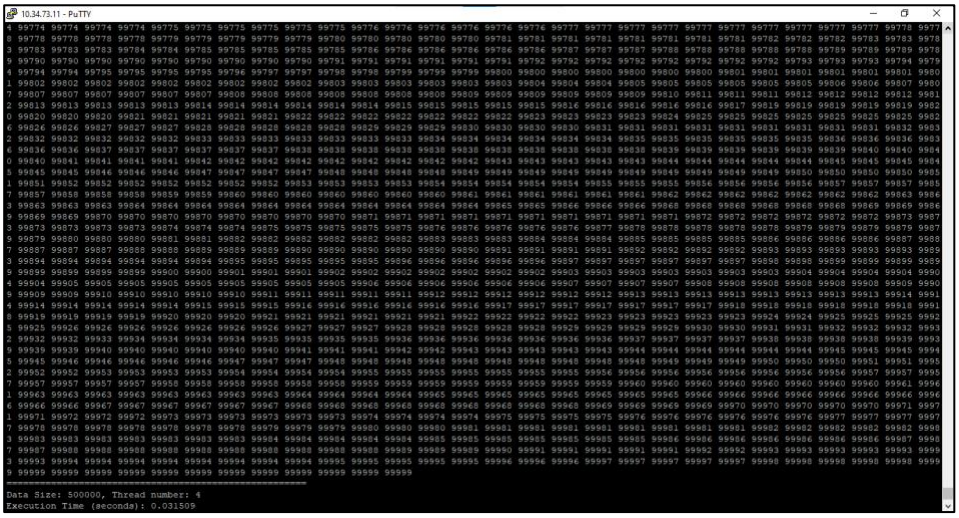
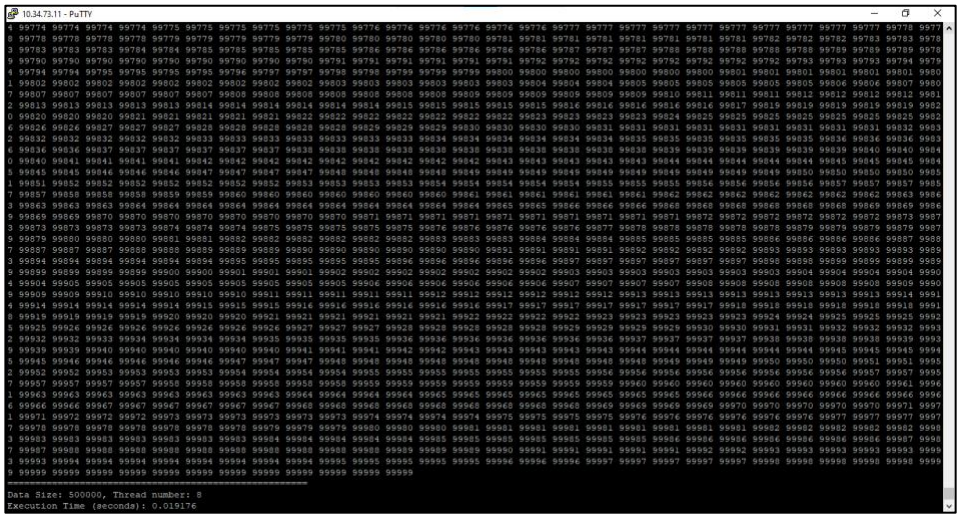
Testing result - Screenshot

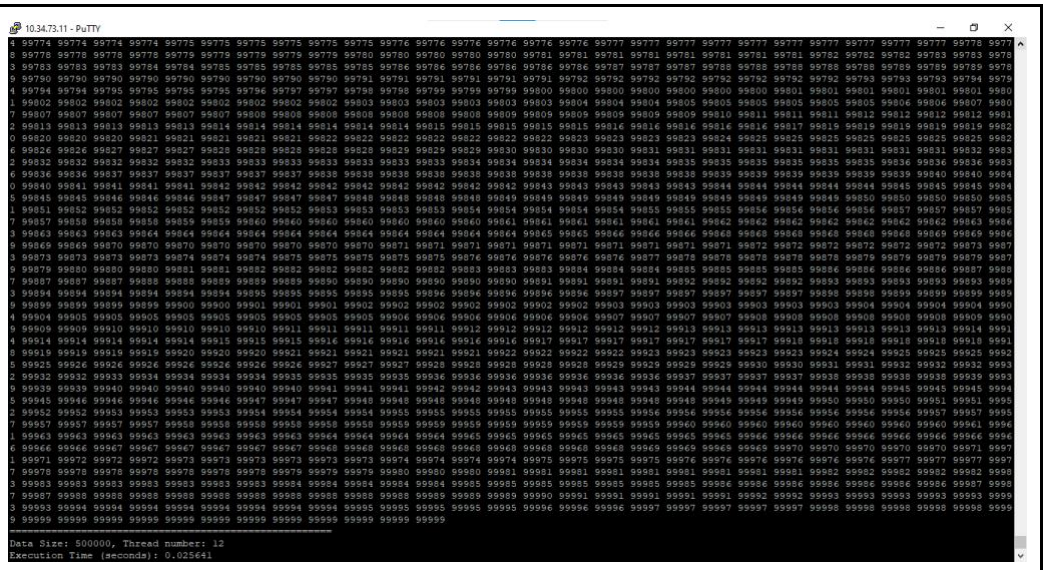
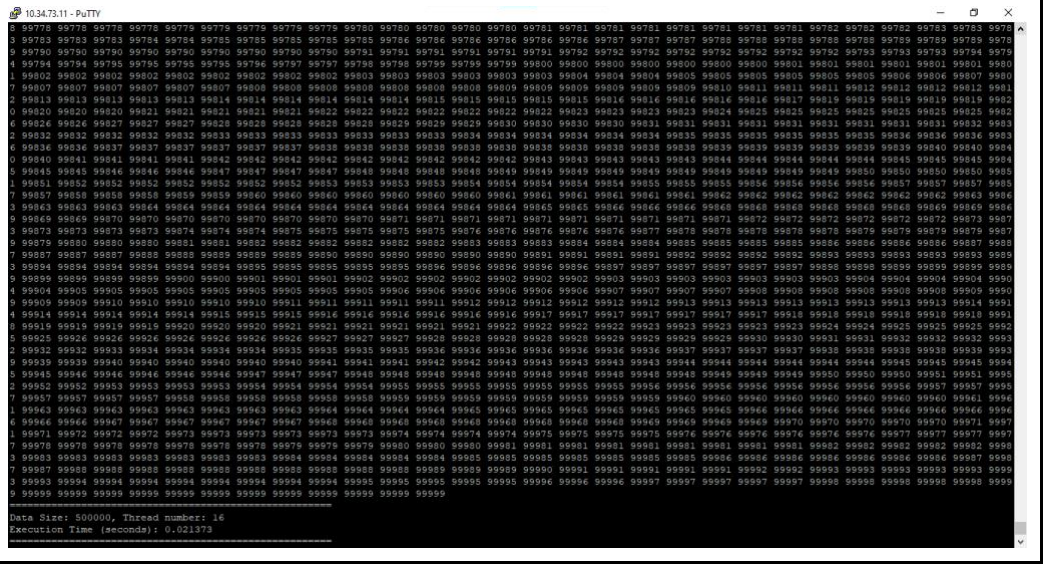
Data set 10,000 random integer numbers	
P = 1	
P = 4	
P = 8	

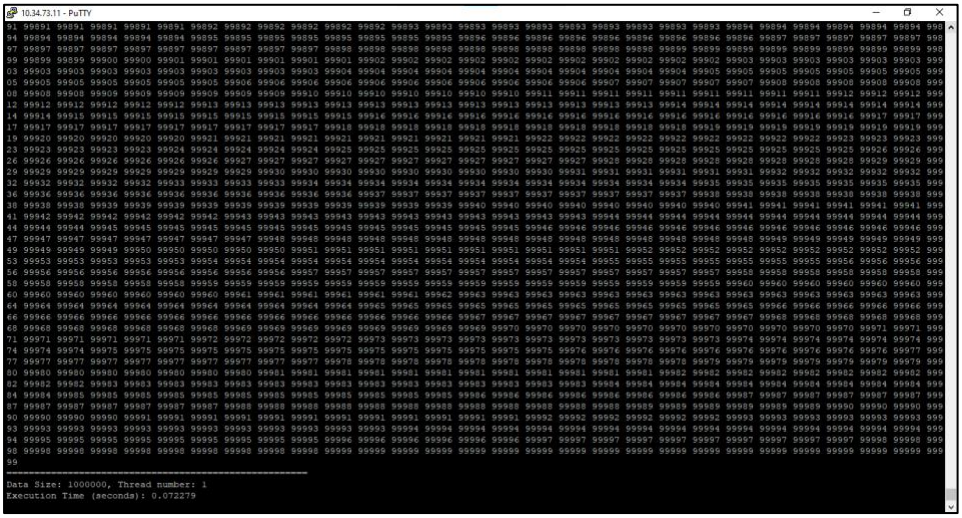
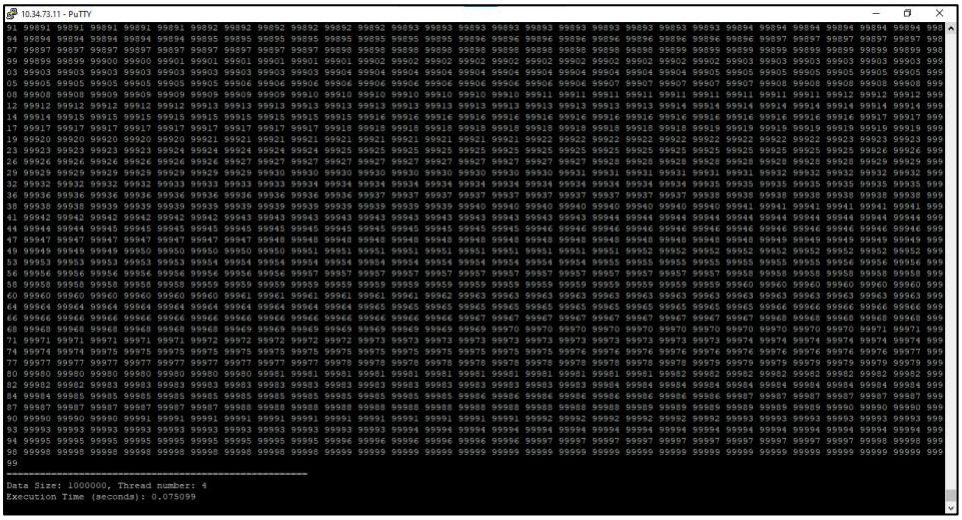
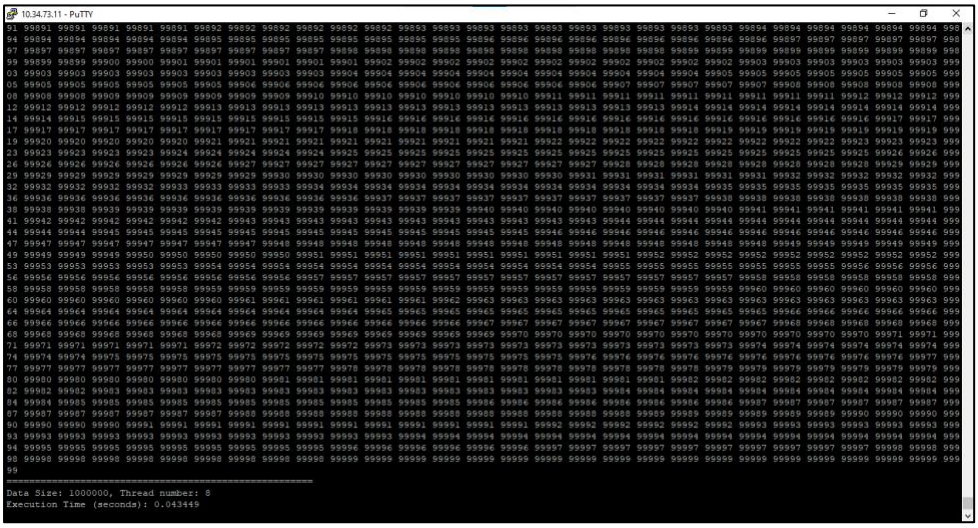
P = 12	 10.34.73.11 - PuTTY Data Size: 10000, Thread number: 12 Execution Time (seconds): 0.003283
P = 16	 10.34.73.11 - PuTTY Data Size: 10000, Thread number: 16 Execution Time (seconds): 0.003268

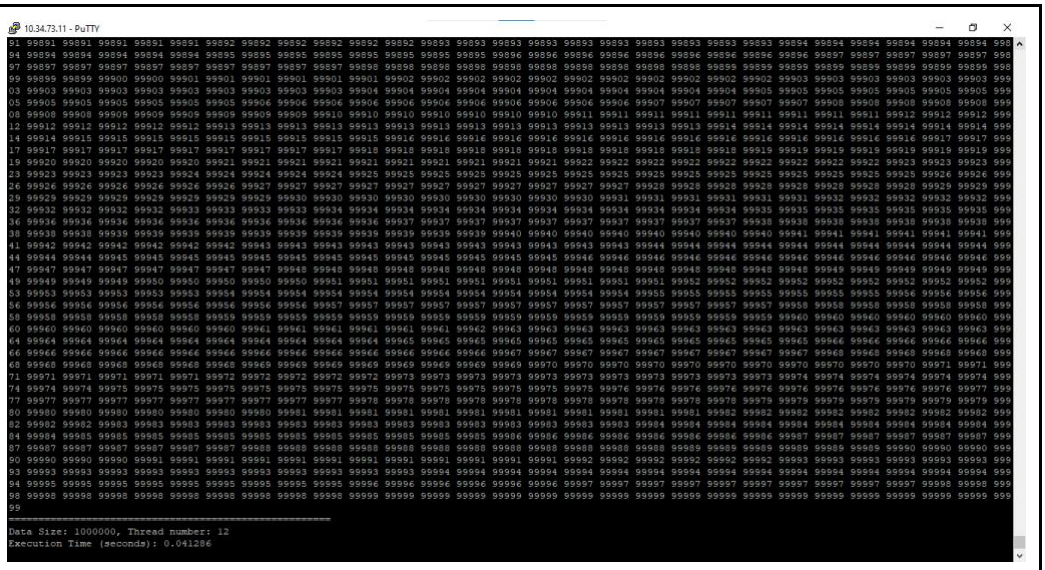
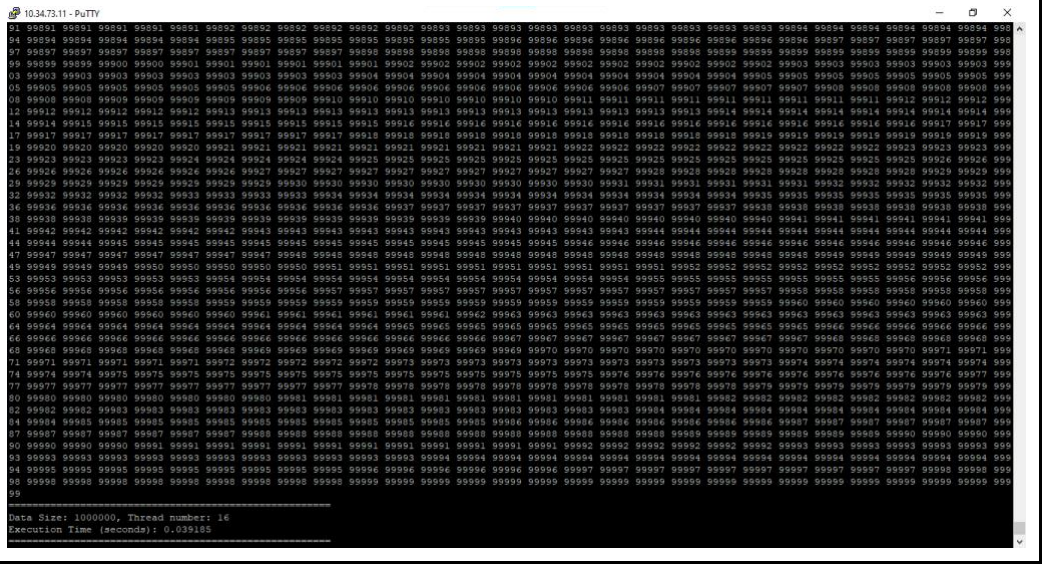
Data set 100,000 random integer numbers	
P = 1	
P = 4	
P = 8	

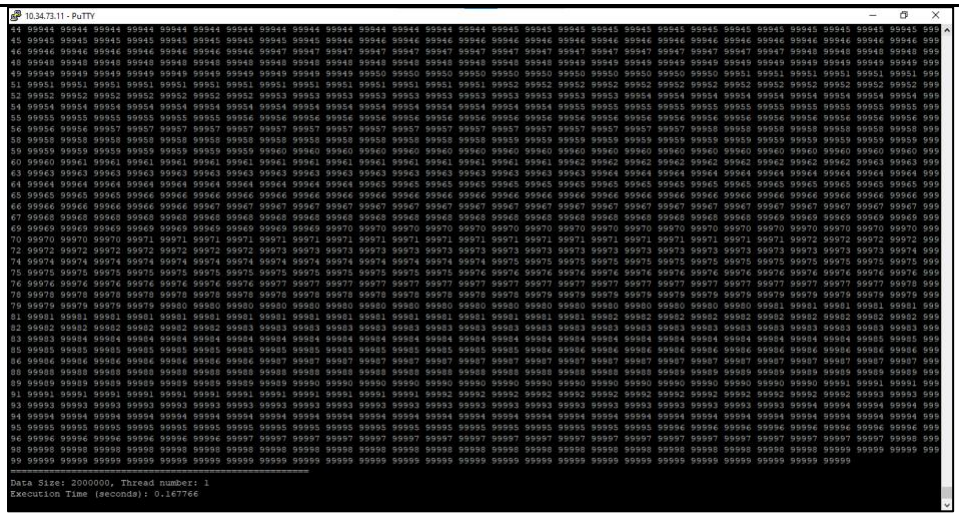
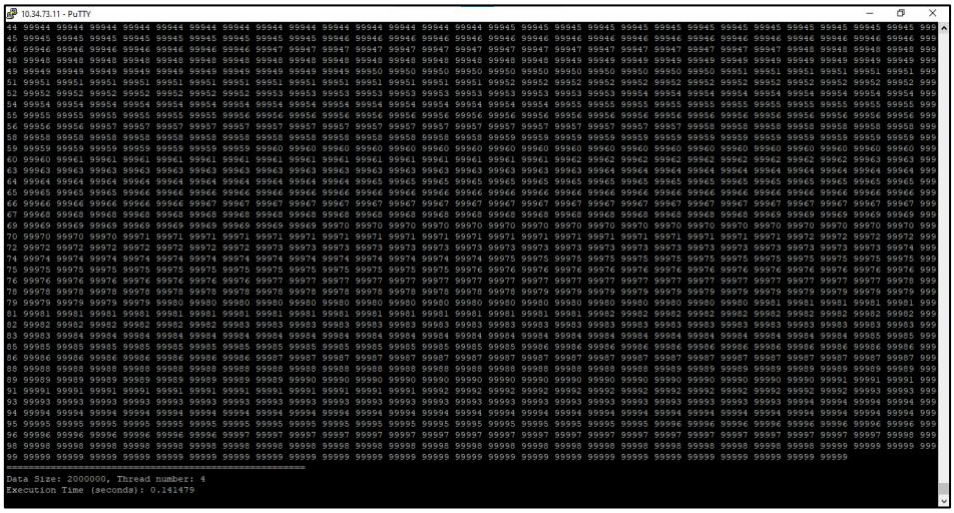
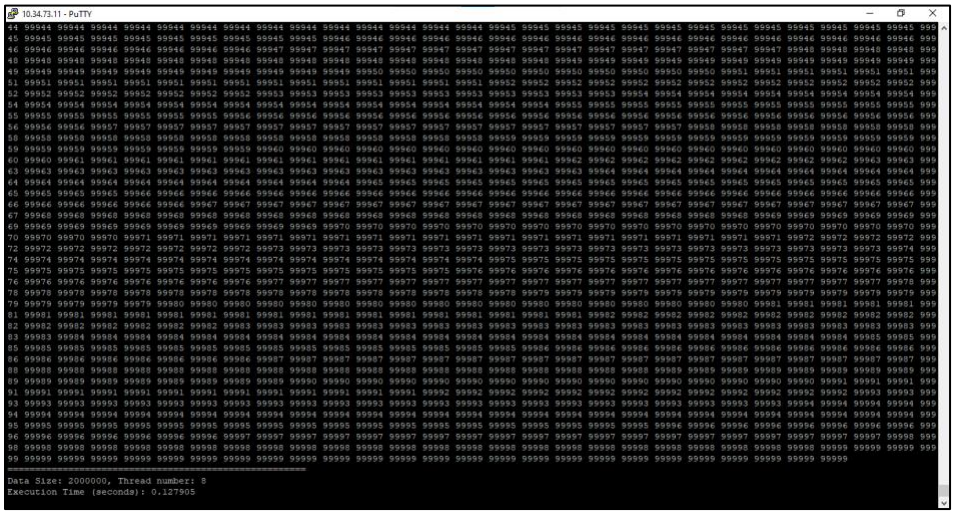
P = 12	 10.34.73.11 - PuTTY Data Size: 100000, Thread number: 12 Execution Time (seconds): 0.005365
P = 16	 10.34.73.11 - PuTTY Data Size: 100000, Thread number: 16 Execution Time (seconds): 0.003052

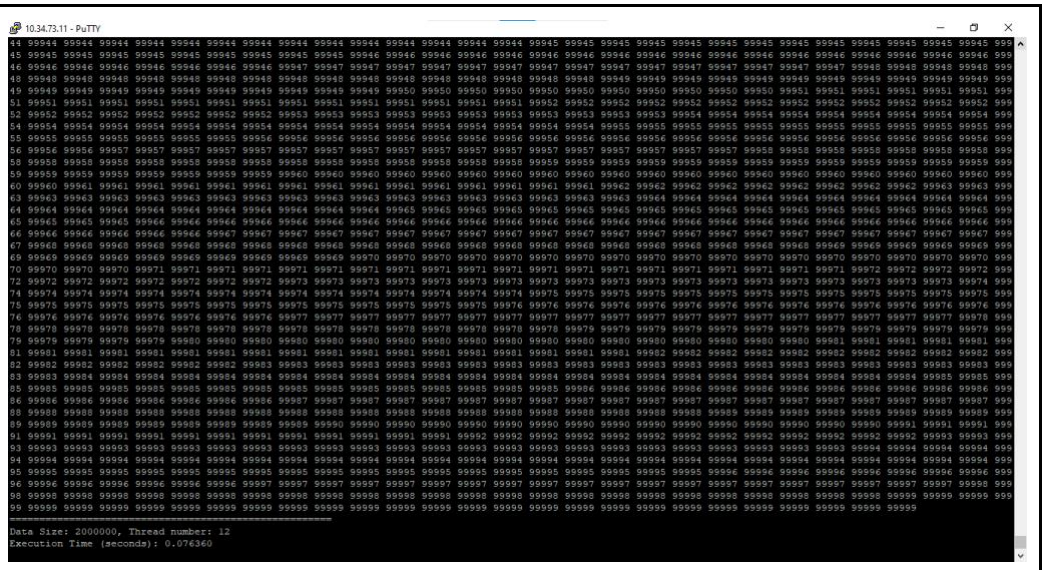
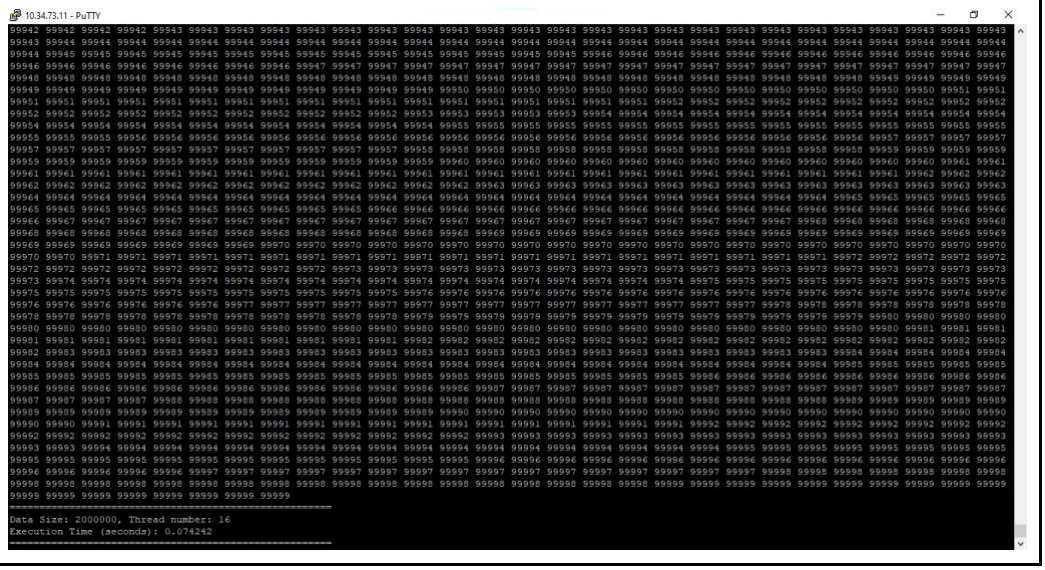
Data set 500,000 random integer numbers	
P = 1	
P = 4	
P = 8	

P = 12	 10.34.73.11 - PuTTY Data Size: 500000, Thread number: 12 Execution Time (seconds): 0.025641
P = 16	 10.34.73.11 - PuTTY Data Size: 500000, Thread number: 16 Execution Time (seconds): 0.021373

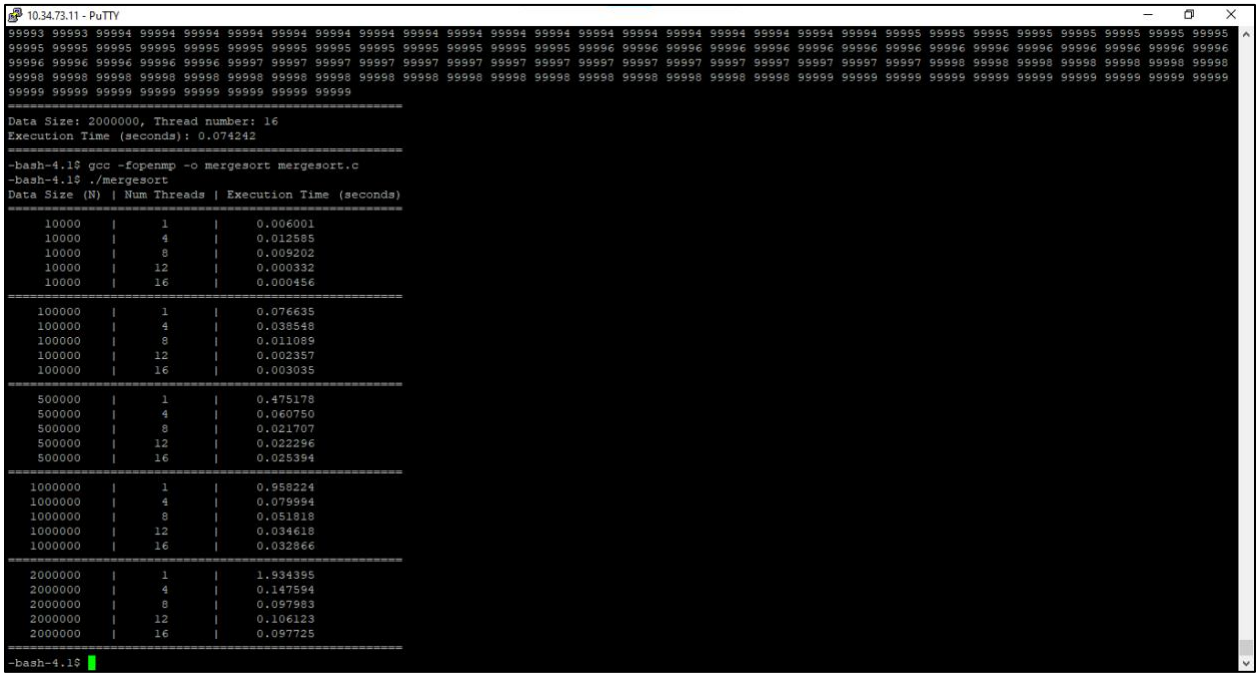
Data set 1,000,000 random integer numbers	
P = 1	
P = 4	
P = 8	

P = 12	 10.34.73.11 - PuTTY Data Size: 1000000, Thread number: 12 Execution Time (seconds): 0.041286
P = 16	 10.34.73.11 - PuTTY Data Size: 1000000, Thread number: 16 Execution Time (seconds): 0.039185

Data set 2,000,000 random integer numbers	
P = 1	
P = 4	
P = 8	

P = 12	 10.34.73.11 - PuTTY Data Size: 2000000, Thread number: 12 Execution Time (seconds): 0.076360
P = 16	 10.34.73.11 - PuTTY Data Size: 2000000, Thread number: 16 Execution Time (seconds): 0.074242

Testing result table – Screenshot on PuTTY



Testing result table

- **Data set 10,000 random integer numbers**

N of Threads / Trial Set	1	4	8	12	16
Set 1	0.003106	0.007394	0.000849	0.000484	0.001022
Set 2	0.003102	0.000300	0.005816	0.000720	0.000637
Set 3	0.003169	0.000622	0.000455	0.000608	0.000671
Set 4	0.003018	0.000275	0.006441	0.000436	0.000671
Set 5	0.003013	0.000335	0.000515	0.000401	0.000447
Average	0.0030816	0.0017852	0.0028152	0.0005298	0.0006896
Speed Up	1	1.726193144	1.09462916	5.81653454	4.46867749

- **Data set 100,000 random integer numbers**

N of Threads / Trial Set	1	4	8	12	16
Set 1	0.042114	0.021525	0.011907	0.003952	0.002551
Set 2	0.046678	0.002902	0.002162	0.002595	0.005552
Set 3	0.054252	0.019436	0.002459	0.003482	0.002119
Set 4	0.044777	0.003131	0.004087	0.010883	0.002307
Set 5	0.059678	0.003393	0.001643	0.002580	0.002750
Average	0.0494998	0.0100774	0.0044516	0.0046984	0.0030558
Speed Up	1	4.911961419	11.1195525	10.5354589	16.1986387

Testing result table

- **Data set 500,000 random integer numbers**

N of Threads / Trial Set	1	4	8	12	16
Set 1	0.253251	0.009861	0.016426	0.027955	0.009160
Set 2	0.243190	0.025369	0.020490	0.10222	0.017037
Set 3	0.239614	0.009536	0.009227	0.016659	0.027199
Set 4	0.255801	0.011405	0.031367	0.010646	0.009637
Set 5	0.261955	0.010347	0.030555	0.011877	0.020454
Average	0.2507622	0.0133036	0.021613	0.0338714	0.0166974
Speed Up	1	18.84919871	11.6023782	7.40336095	15.0180387

- **Data set 1,000,000 random integer numbers**

N of Threads / Trial Set	1	4	8	12	16
Set 1	0.479794	0.052546	0.030082	0.034523	0.019378
Set 2	0.467432	0.060649	0.041398	0.032933	0.035166
Set 3	0.476305	0.051676	0.047858	0.036101	0.031697
Set 4	0.470280	0.067127	0.055691	0.035658	0.023004
Set 5	0.549390	0.022881	0.036992	0.026048	0.028851
Average	0.4886402	0.0509758	0.0424042	0.0330526	0.0276192
Speed Up	1	9.585728915	11.5233916	14.7837144	17.6920476

Testing result table

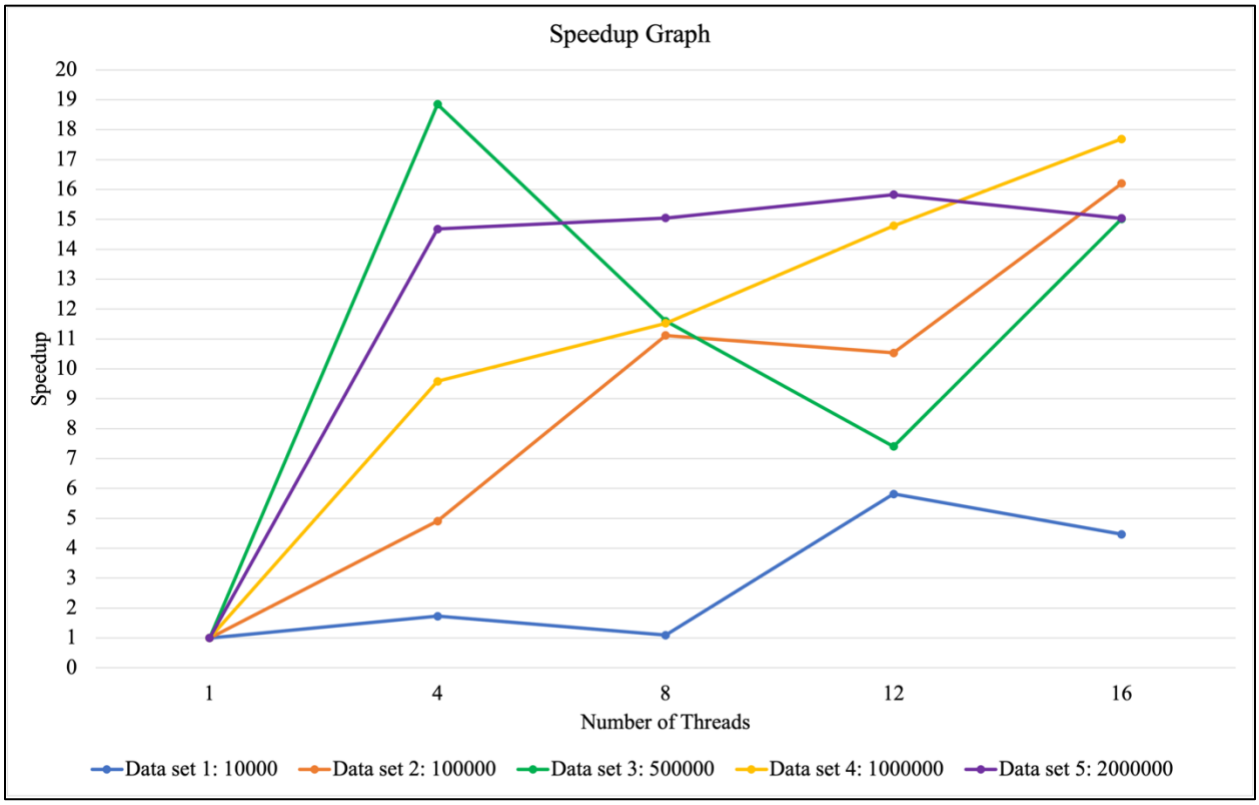
- **Data set 2,000,000 random integer numbers**

N of Threads / Trial Set	1	4	8	12	16
Set 1	1.022677	0.072124	0.059078	0.067891	0.072779
Set 2	1.008235	0.076407	0.057658	0.054718	0.063926
Set 3	0.986500	0.066537	0.089172	0.068558	0.088041
Set 4	1.073811	0.059825	0.053130	0.083254	0.066211
Set 5	1.053302	0.075541	0.082894	0.0505844	0.051266
Average	1.028905	0.0700868	0.0683864	0.06500108	0.0684446
Speed Up	1	14.68043911	15.0454623	15.8290447	15.0326688

Speedup Comparison Table

Speedup					
P	1	4	8	12	16
Data set 1: 10000	1	1.726193144	1.09462916	5.81653454	4.46867749
Data set 2: 100000	1	4.911961419	11.1195525	10.5354589	16.1986387
Data set 3: 500000	1	18.84919871	11.6023782	7.40336095	15.0180387
Data set 4: 1000000	1	9.585728915	11.5233916	14.7837144	17.6920476
Data set 5: 2000000	1	14.68043911	15.0454623	15.8290447	15.0326688

Speed Up Graph Comparison



The graph shows that the speedup generally increases as the number of compute threads increases, but that the rate of speedup decreases as the number of compute threads increases. The graph also shows that the speedup varies depending on the size of the data set. For example, Data Set 5 shows the highest speedup, while Data Set 1 shows the lowest speedup. This is because larger data sets are typically more amenable to parallelization, since they can be divided into more independent tasks.