Group Assignment:
Parallel Sorting Algorithms

Topic: Parallel Bucketsort via OpenMP

By

Mr. Waris Damkham 6388014
Miss Pattanan Korkiattrakool 6388022
Mr. Naphat Sookjitsumran 6388059

Submitted to

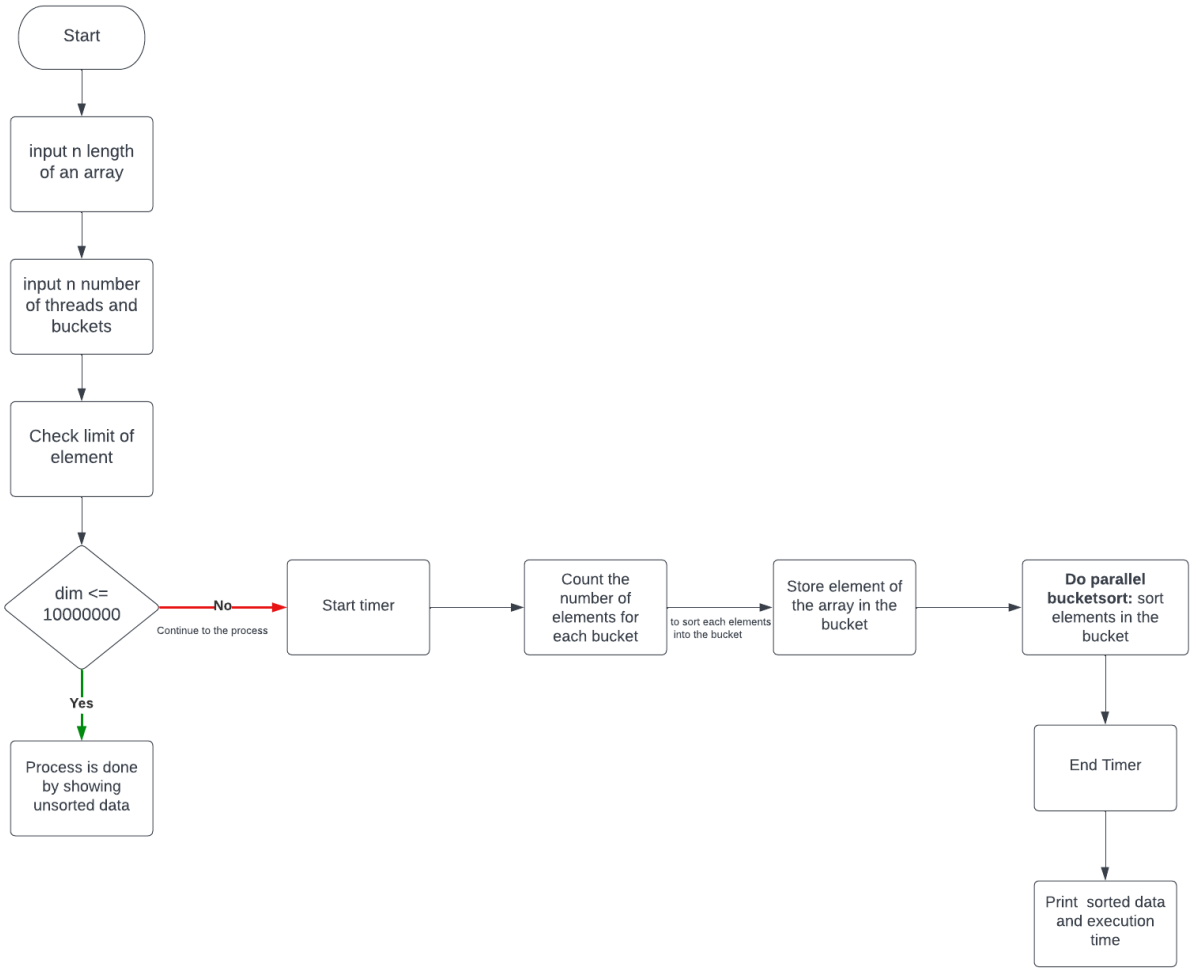
Assoc. Prof. Dr. Sudsanguan Ngamsuriyaroj

A Report Submitted in Partial Fulfillment of the Requirements for
ITCS443_Parallel and Distributed Systems
Faculty of Information and Communication Technology Mahidol University
1/2022

Table of Contents

Explanation of our program as flowchart (Parallel Bucketsort via OpenMP)

- Flowchart explained how the program process the result to be the bucket sort from the beginning to the end.



Source code

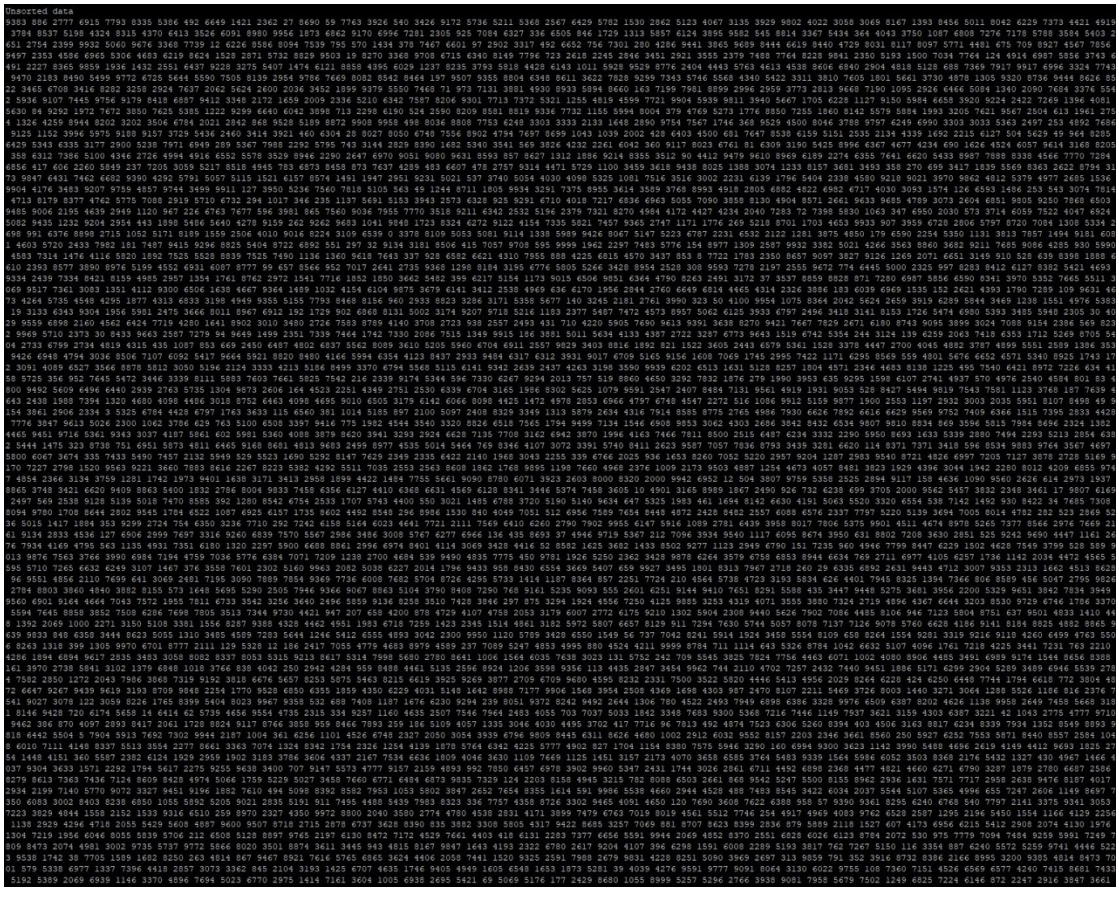
```c
#include <omp.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#define max_threads 100000
struct bucket {
    int n_elem;
    int index; // [start : n_elem)
    int start; //starting point in B array
};
int cmpfunc (const void * a, const void * b)
{
 return ( *(int*)a - *(int*)b );
}
int main(int argc, char *argv[]) {
    int *A, *B, *temp;
    int dim, n_buckets, i, w, limit, num_threads, workload, b_index;
    struct bucket* buckets; //array of buckets
    double t1; // Timing variable
    float total; //total time
    char operation;
    int ss ;
    printf("Welcome to Parallel Bucketsort via OpenMP \n");
    printf("Length of array to sort \n");
    printf("1. 10,000 arrays \n");
    printf("2. 100,000 arrays \n");
    printf("3. 1,000,000 arrays \n");
    printf("Enter length of array (1., 2., 3.): ");
    scanf("%c", &operation);
    switch(operation)
    {
        case '1':
            dim = 10000;
            break;
        case '2':
            dim = 100000;
            break;
        case '3':
            dim = 1000000;
            break;
        // operator doesn't match any case constant +, -, *, /
        default:
            printf("Error! operator is not correct");
            return -1;
    }
    printf("-------------------------------------------------------\n");
    printf("Number of buckets / threads \n");
    printf("1. 1 thread \n");
    printf("2. 4 threads \n");
    printf("3. 8 threads \n");
    printf("4. 12 threads \n");
    printf("5. 16 threads \n");
    printf("Enter number of threads and buckets (1., 2., 3., 4., 5.): ");
    scanf("%d", &ss);
    switch(ss)
    {
        case 1:
            n_buckets = 1;
            break;
        case 2:
            n_buckets = 4;
            break;
        case 3:
            n_buckets = 8;
            break;
        case 4:
            n_buckets = 12;
            break;
        case 5:
```

```
            n_buckets = 16;
            break;
        // operator doesn't match any case constant +, -, *, /
        default:
            printf("Error! operator is not correct");
            return -1;
    }
    printf("-------------------------------------------------------\n");
        int *tmp;
    //global buckets
    int global_n_elem[n_buckets]; //number of elements in each bucket
    int global_starting_position[n_buckets]; //starting position in A for each bucket
    memset(global_n_elem, 0, sizeof(int)*n_buckets);
    memset(global_starting_position, 0, sizeof(int)*n_buckets);
    num_threads = n_buckets;
    omp_set_num_threads(num_threads);
    limit = dim;
    w = limit/n_buckets;
    A = (int *) malloc(sizeof(int)*dim);
    B = (int *) malloc(sizeof(int)*dim);
    for(i=0;i<dim;i++) {
            A[i] = random() % limit;
        }
        if (dim <= 10000000) {
        printf("Unsorted data \n");
        for(i=0;i<dim;i++) {
            printf("%d ",A[i]);
        }
        printf("\n");
    }
    //local buckets, n_buckets for each thread
    buckets = (struct bucket *) calloc(n_buckets*num_threads, sizeof(struct bucket));
// ****************************
// Starting the main algorithm
// ****************************
    t1 = omp_get_wtime();
#pragma omp parallel
{
    num_threads = omp_get_num_threads();
    int j,k;
    int local_index; // [0 : n_buckets)
    int real_bucket_index; // [0 : n_buckets * num_threads)
    int my_id = omp_get_thread_num();
    workload = dim/num_threads;
        int prevoius_index;
  #pragma omp for private(i,local_index)
    for (i=0; i< dim;i++){
        local_index = A[i]/w;
        if (local_index > n_buckets-1)
                local_index = n_buckets-1;
        real_bucket_index = local_index + my_id*n_buckets;
        buckets[real_bucket_index].n_elem++;
        }
        int local_sum=0;
        for (j=my_id; j< n_buckets*num_threads; j=j+num_threads){
                local_sum += buckets[j].n_elem;
        }
        global_n_elem[my_id]=local_sum;
        #pragma omp barrier
        #pragma omp master
    {
         for (j=1; j<n_buckets; j++){
                        global_starting_position[j] = global_starting_position[j-1] + global_n_elem[j-1];
                                buckets[j].start = buckets[j-1].start + global_n_elem[j-1];
                                buckets[j].index = buckets[j-1].index + global_n_elem[j-1];
                }
        }
        #pragma omp barrier
        for (j=my_id+n_buckets; j< n_buckets*num_threads; j=j+num_threads){
                int prevoius_index = j-n_buckets;
                        buckets[j].start = buckets[prevoius_index].start + buckets[prevoius_index].n_elem;
                        buckets[j].index = buckets[prevoius_index].index + buckets[prevoius_index].n_elem;
        }
        #pragma omp barrier
    #pragma omp for private(i, b_index)
    for (i=0; i< dim ;i++){
        j = A[i]/w;
        if (j > n_buckets -1)
                j = n_buckets-1;
        k = j + my_id*n_buckets;
        b_index = buckets[k].index++;
        B[b_index] = A[i];
    }
#pragma omp for private(i)
    for(i=0; i<n_buckets; i++)
        qsort(B+global_starting_position[i], global_n_elem[i], sizeof(int), cmpfunc);
}
   total = omp_get_wtime() - t1;
        tmp = A;
        A = B;
        B = tmp;
   if (dim <= 1000000) {
        printf("Sorted Data \n");
        for(i=0;i<dim;i++) {
            printf("%d ",A[i]);
        }
        printf("\n");
    }
    printf("Sorting %d elements took %f seconds\n", dim,total);
    int sorted = 1;
    for(i=0;i<dim-1;i++) {
        if (A[i] > A[i+1])
            sorted = 0;
    }
    if (!sorted)
        printf("The data is not sorted!!!\n");
}
```

Testing result with capture screen shots

- Running on the ICT cluster system and the output of unsorted data.

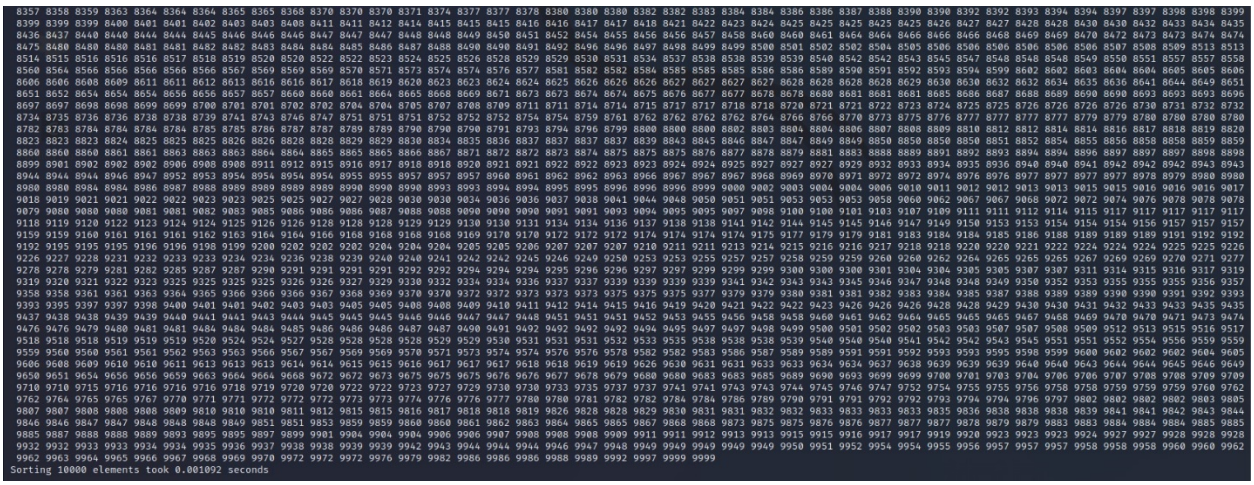
```
[u6388014@cluster ~]$ ./para
Welcome to Parallel Bucketsort via OpenMP
Length of array to sort
1. 10,000 arrays
2. 100,000 arrays
3. 1,000,000 arrays
Enter length of array (1., 2., 3.): 1
---------------------------------------------------------
Number of buckets / threads
1. 1 thread
2. 4 threads
3. 8 threads
4. 12 threads
5. 16 threads
Enter number of threads and buckets (1., 2., 3., 4., 5.): 1
---------------------------------------------------------
```



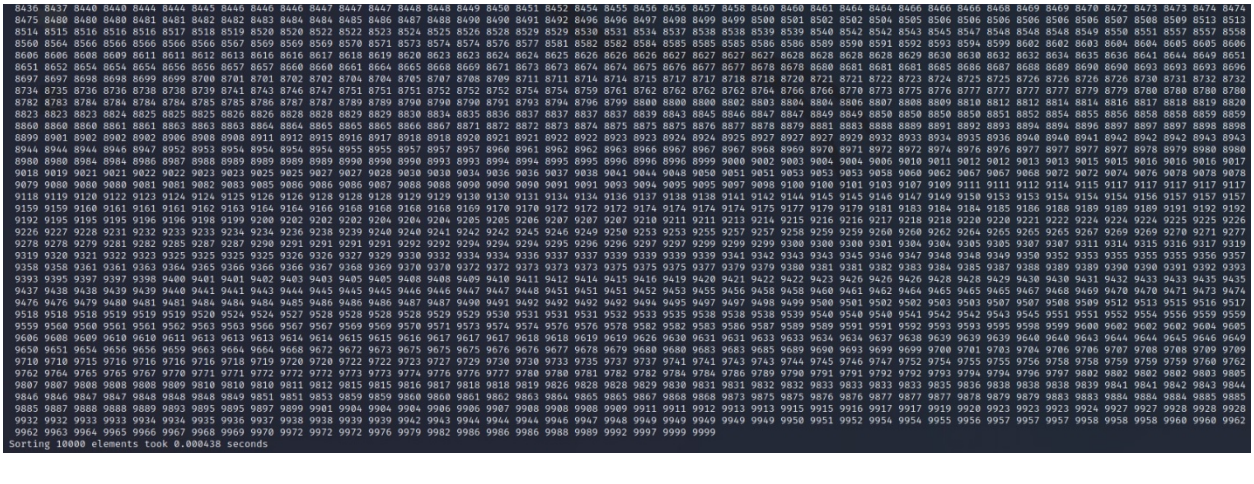
Testing result

1. Generate 10,000 random integer numbers with 1,4,8,12 and 16 threads.

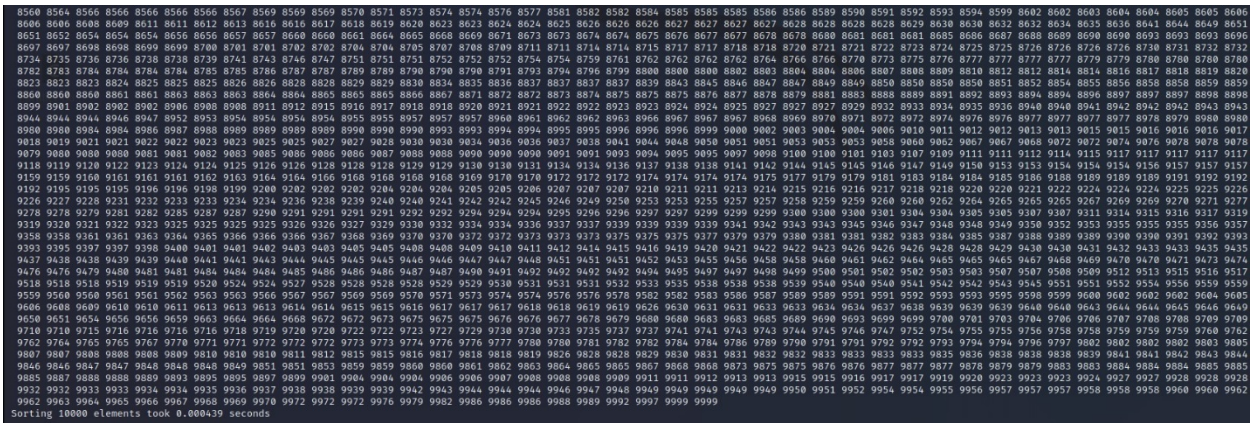
- **1 thread** with 0.001092 seconds of the execution time.



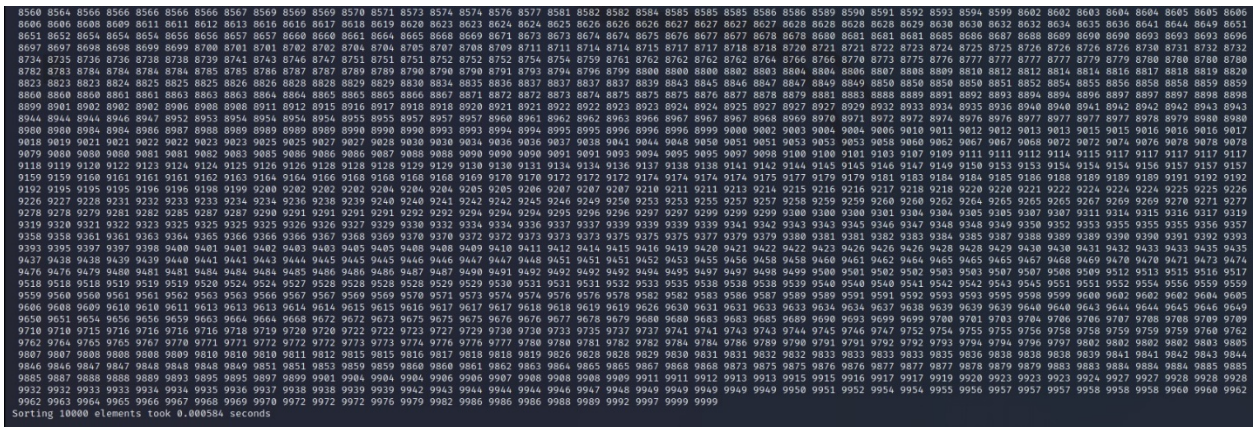
- **4 threads** with 0.000438 seconds of the execution time.



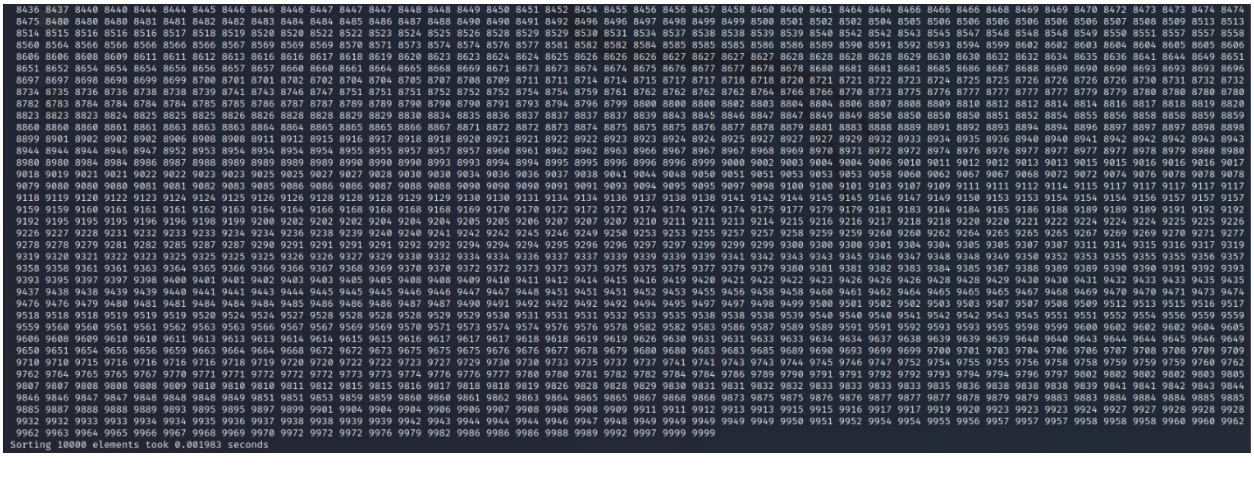
- **8 threads** with 0.000439 seconds of the execution time.



- **12 threads** with 0.000584 seconds of the execution time.

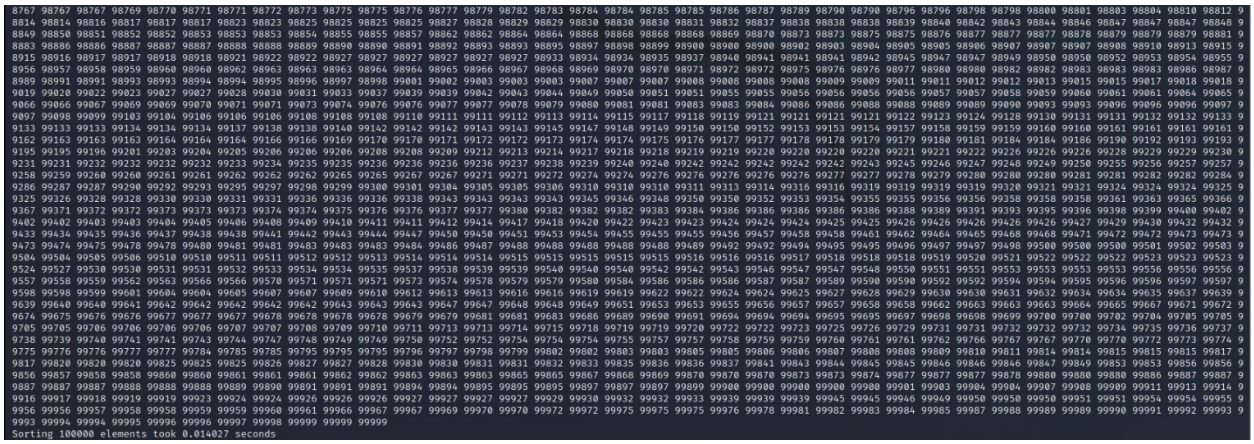


- **16 threads** with 0.001983 seconds of the execution time.

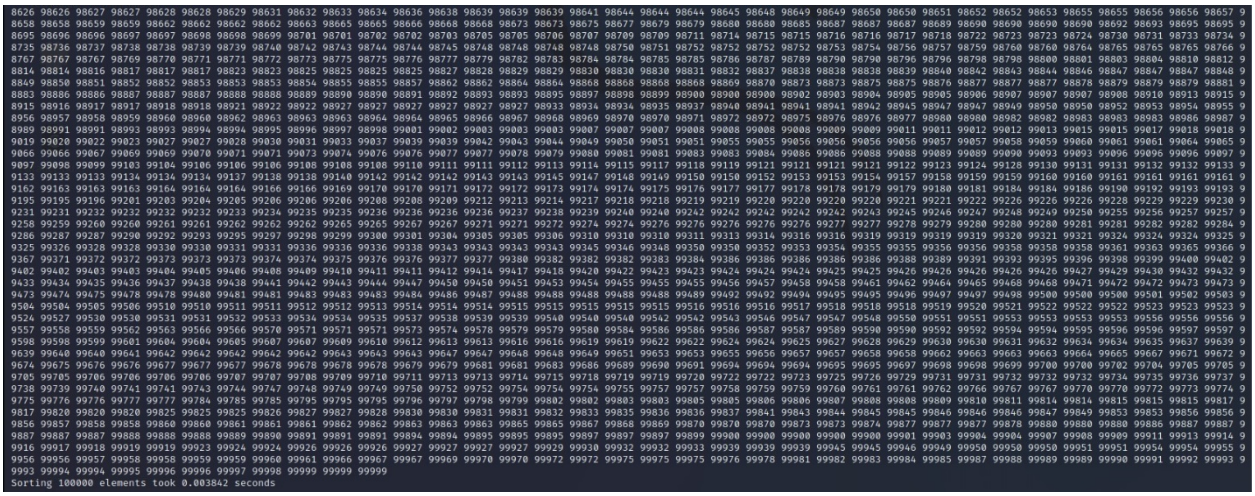


2. Generate 100,000 random integer numbers with 1,4,8,12 and 16 threads.

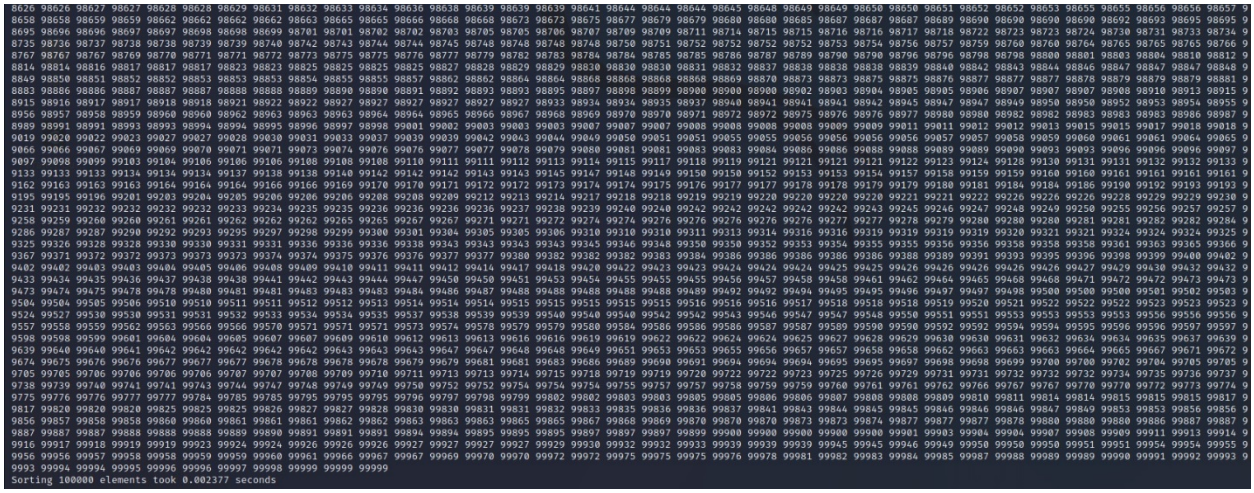
- **1 thread** with 0.014027 seconds of the execution time.



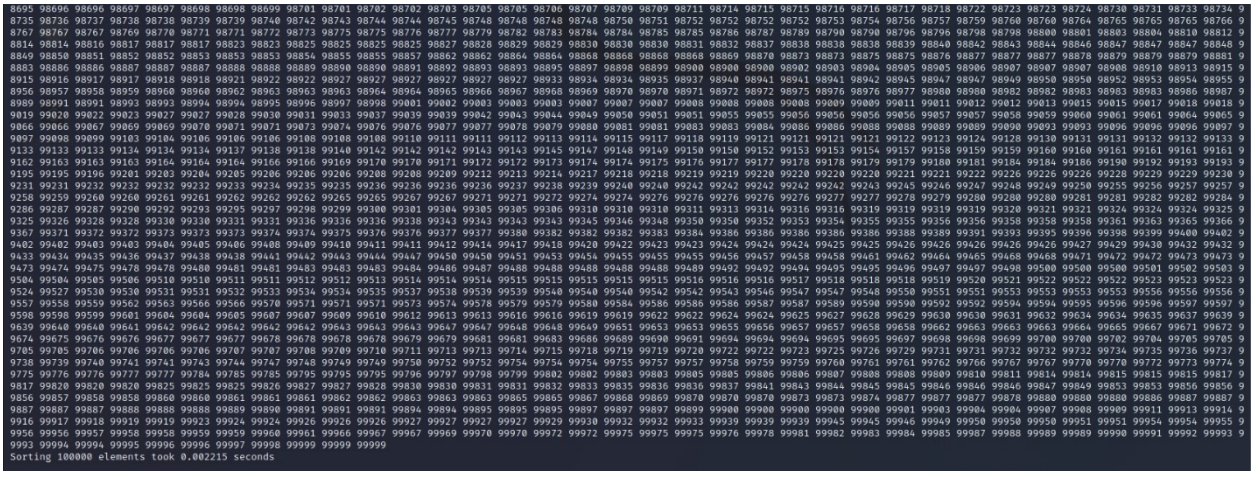
- **4 threads** with 0.003842 seconds of the execution time.



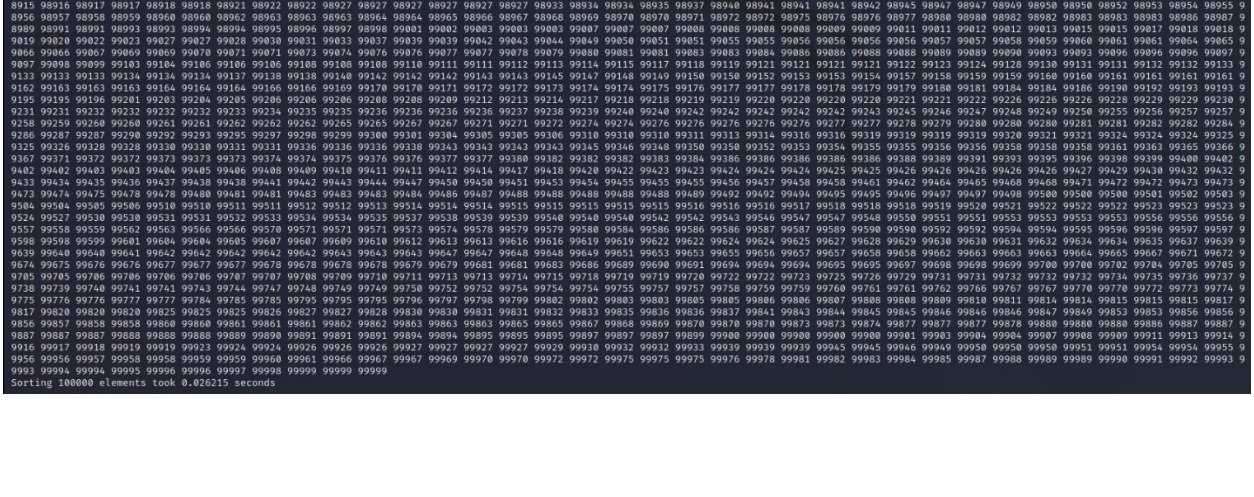
- **8 threads** with 0.002377 seconds of the execution time.



- **12 threads** with 0.002215 seconds of the execution time.

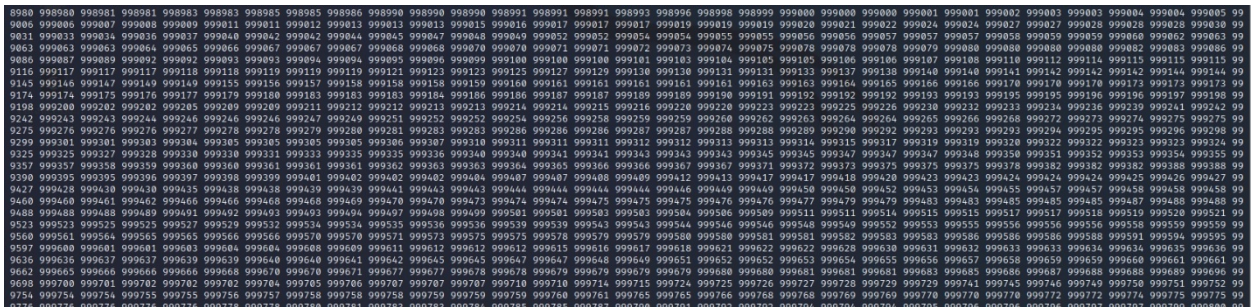


- **16 threads** with 0.026215 seconds of the execution time.

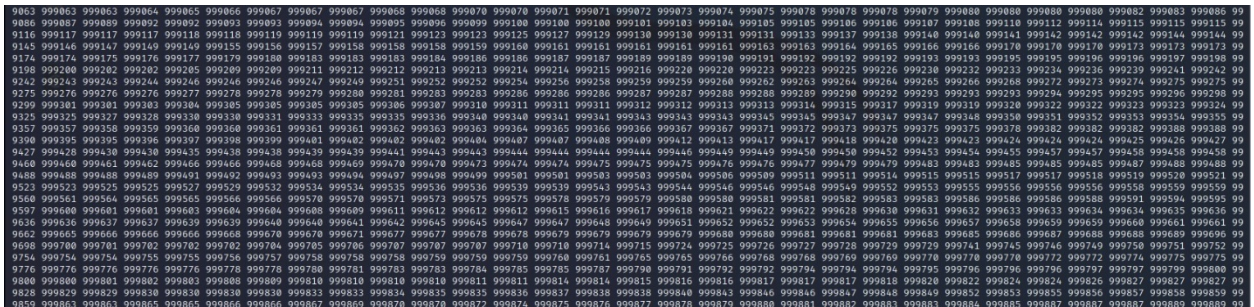


3. Generate 1,000,000 random integer numbers with 1,4,8,12 and 16 threads.

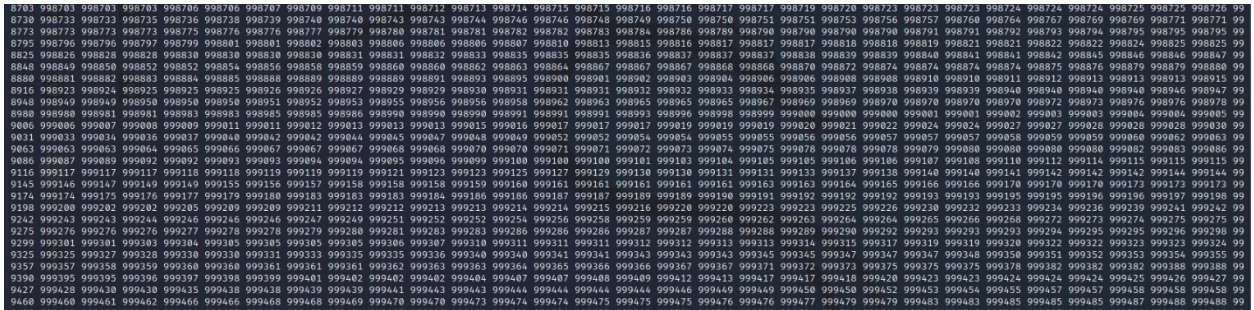
- **1 thread** with 0.149628 seconds of the execution time.



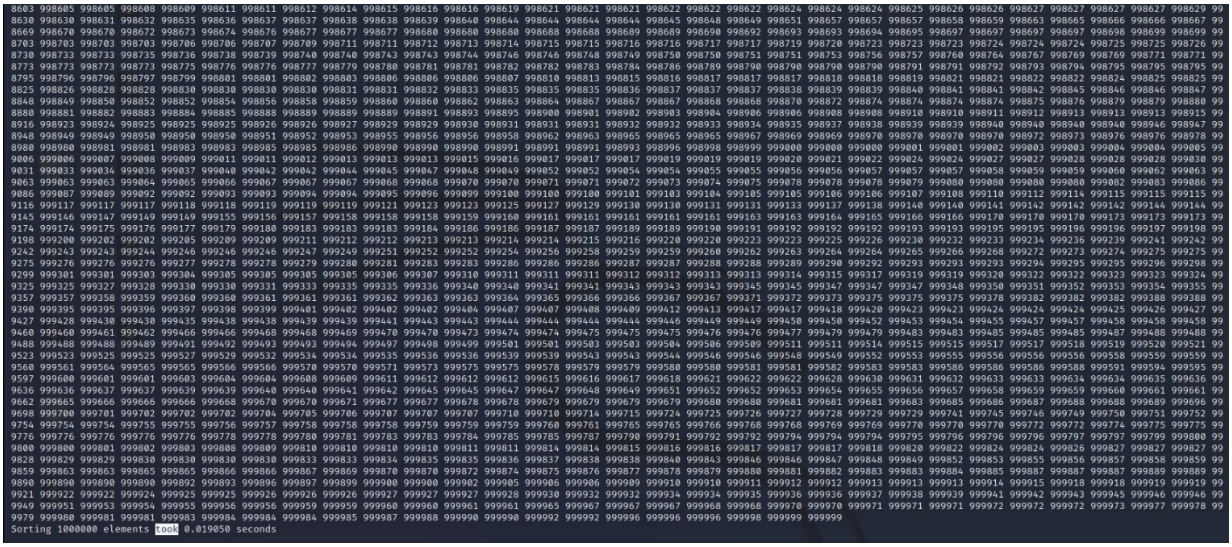
- **4 threads** with 0.046378 seconds of the execution time.



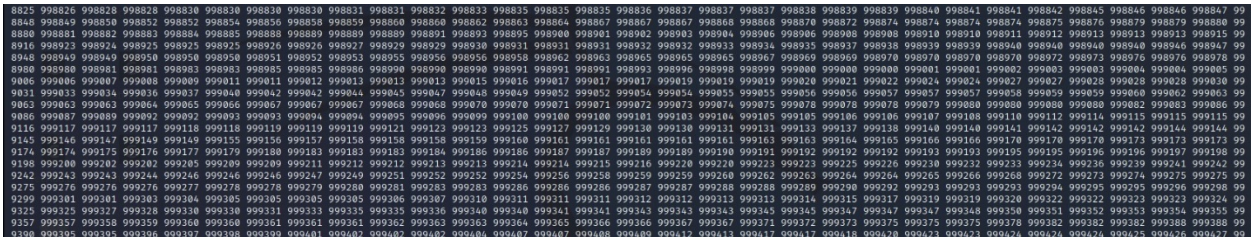
- **8 threads** with 0.024327 seconds of the execution time.



- **12 threads** with 0.019150 seconds of the execution time.

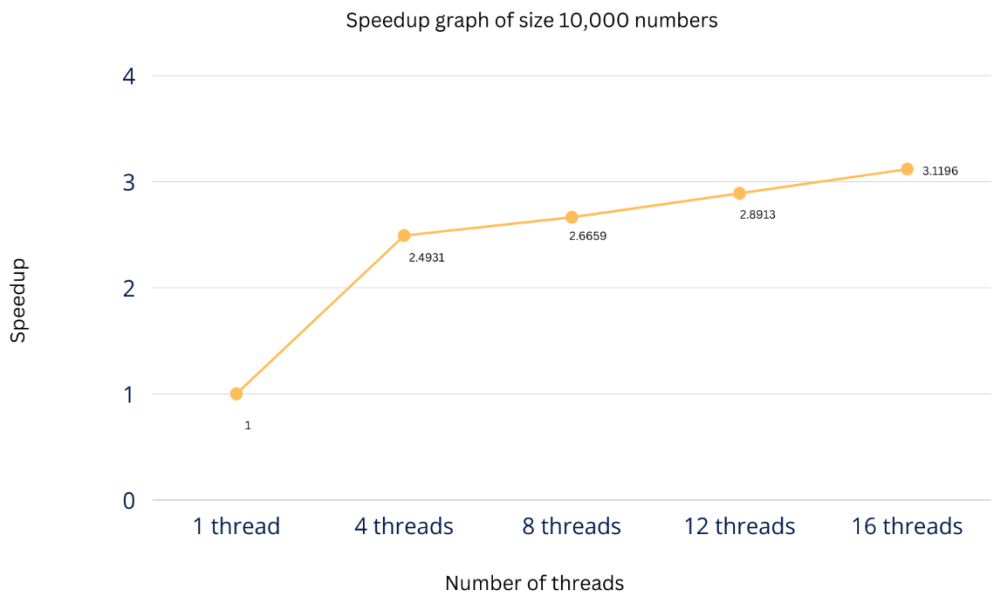


- **16 threads** with 0.075496 seconds of the execution time.

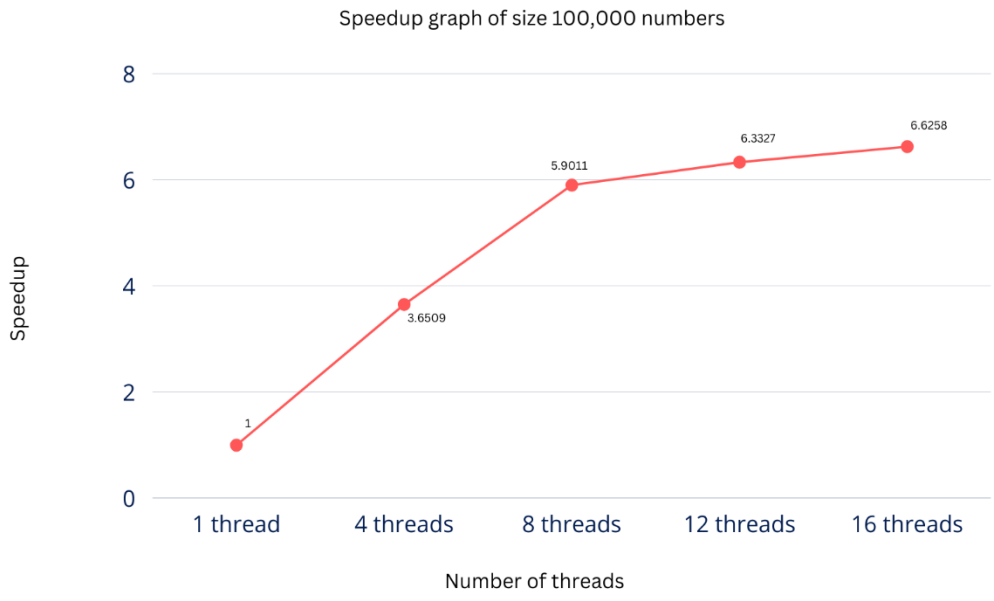


Speedup graph

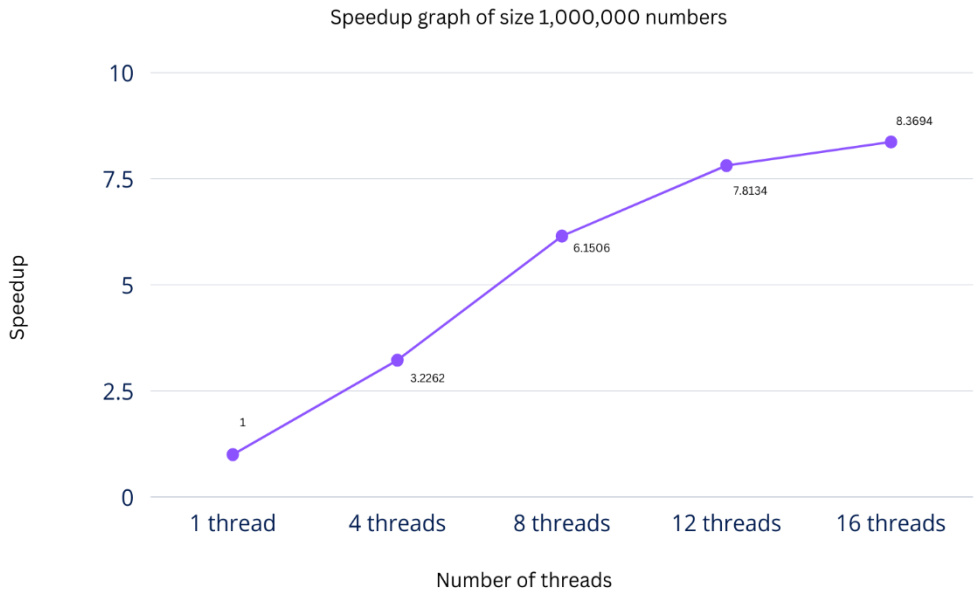
- Speedup graph of size 10,000 numbers



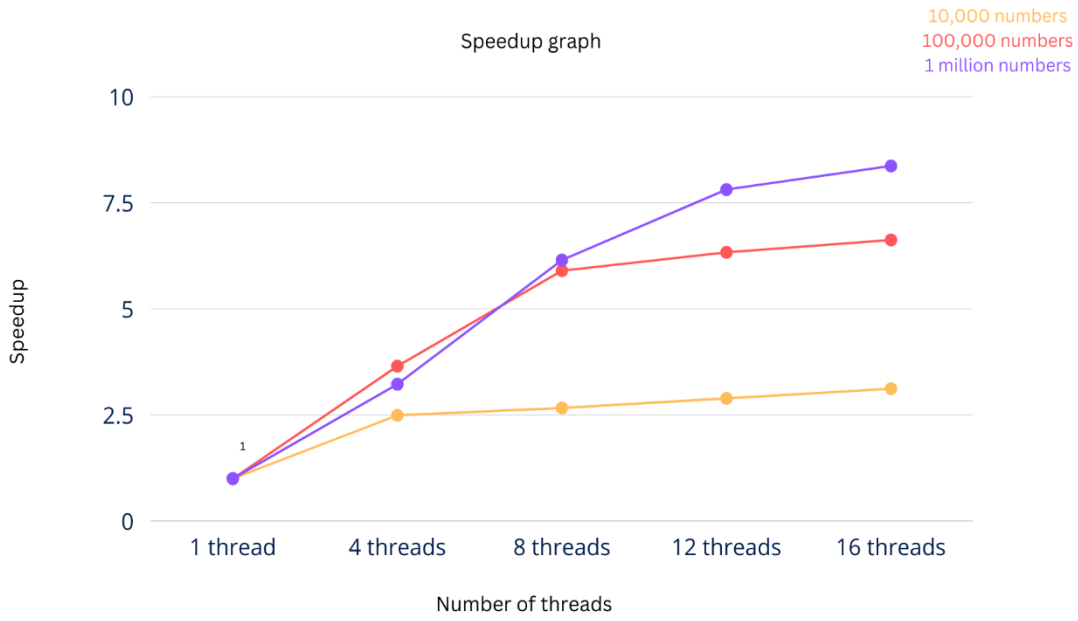
- Speedup graph of size 100,000 numbers



- Speedup graph of size 1 million numbers



- Speedup graph shows every result of generate numbers



Explanation of the result

10,000 numbers		100,000 numbers		1 million numbers	
Sequential	Parallel	Sequential	Parallel	Sequential	Parallel
0.001092	0.001092	0.014027	0.014027	0.149628	0.149628
0.001092	0.000438	0.014027	0.003842	0.149628	0.046378
0.001092	0.000439	0.014027	0.002377	0.149628	0.024327
0.001092	0.000584	0.014027	0.002215	0.149628	0.019150
0.001092	0.000376	0.014027	0.002117	0.149628	0.017877

Number of threads	Speedup graph		
	10,000 numbers	100,000 numbers	1 million numbers
1	1	1	1
4	2.4931	3.6509	3.2262
8	2.6659	5.9011	6.1506
12	2.8913	6.3327	7.8134
16	3.1196	6.6258	8.3694

- From the results, the speedup time of 1 thread is less than to 4, 8, 12, and 16 threads. Accordingly, these tables show that the execution time of parallel is less than the execution time of sequential. Moreover, after we have run many cases, the result is that sometimes more threads use time than fewer threads because sometimes communication time between processors is more than computation time.