



**Group Assignment:
Parallel Sorting Algorithms**

Topic: Parallel Bucket sort via OpenMP

By

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

Submitted to

Assoc. Prof. Dr. Sudsanguan Ngamsuriyaroj

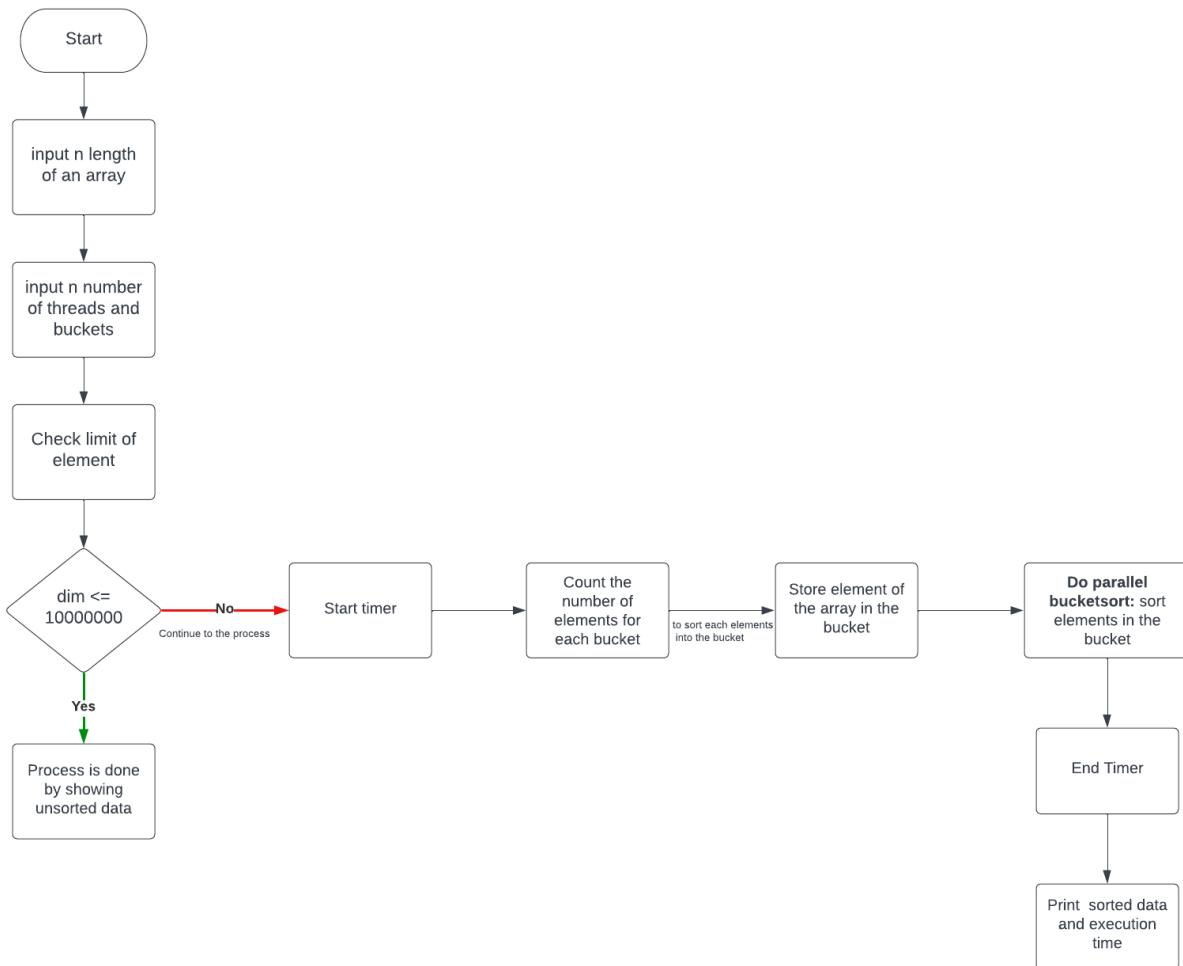
A Report Submitted in Partial Fulfillment of the Requirements for
ITCS443_Parallel and Distributed Systems
Faculty of Information and Communication Technology Mahidol University
1/2022

Table of Contents

<i>Explanation of our program as flowchart (Parallel Bucketsort via OpenMP)</i>	3
<i>Source code</i>	4
<i>Testing result with capture screen shots</i>	6
<i>Testing result</i>	7
<i>Speedup graph</i>	13
<i>Explanation of the result</i>	15

Explanation of our program as flowchart (Parallel Bucket sort via OpenMP)

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



Source code

```
1 #include <omp.h>
2 #include <stdio.h>
3 #include <string.h>
4 #include <stdlib.h>
5 #include <time.h>
6 #define max_threads 100000
7 struct bucket {
8     int n_elem;
9     int index; // [start : n_elem]
10    int start; //starting point in B array
11 };
12 int cmpfunc (const void * a, const void * b)
13 {
14     return ( *(int*)a - *(int*)b );
15 }
16 int main(int argc, char *argv[])
17 {
18     int *A, *B, *temp;
19     int dim, n_buckets, i, w, limit, num_threads, workload, b_index;
20     struct bucket* buckets; //array of buckets
21     double t1; // Timing variable
22     float total; //total time
23     char operation;
24     int ss;
25     printf("Welcome to Parallel Bucket sort via OpenMP \n");
26     printf("Length of array to sort \n");
27     printf("1. 10,000 arrays \n");
28     printf("2. 100,000 arrays \n");
29     printf("3. 1,000,000 arrays \n");
30     printf("Enter length of array (1., 2., 3.): ");
31     scanf("%c", &operation);
32     switch(operation)
33     {
34         case '1':
35             dim = 10000;
36             break;
37         case '2':
38             dim = 100000;
39             break;
40         case '3':
41             dim = 1000000;
42             break;
43         // operator doesn't match any case constant +, -, *, /
44         default:
45             printf("Error! operator is not correct");
46             return -1;
47     }
48     printf("Number of buckets / threads \n");
49     printf("1. 1 thread \n");
50     printf("2. 4 threads \n");
51     printf("3. 8 threads \n");
52     printf("4. 12 threads \n");
53     printf("5. 16 threads \n");
54     printf("Enter number of threads and buckets (1., 2., 3., 4., 5.): ");
55     scanf("%d", &ss);
56     switch(ss)
57     {
58         case 1:
59             n_buckets = 1;
60             break;
61         case 2:
62             n_buckets = 4;
63             break;
64         case 3:
65             n_buckets = 8;
66             break;
67         case 4:
68             n_buckets = 12;
69             break;
70         case 5:
```

```

71     cast .;
72     n_buckets = 16;
73     break;
74   // operator doesn't match any case constant +, -, *, /
75   default:
76     printf("Error! operator is not correct");
77   return -1;
78 }
79 int *tmp;
80 //global buckets
81 int global_n_elem[n_buckets]; //number of elements in each bucket
82 int global_starting_position[n_buckets]; //starting position in A for each bucket
83 memset(global_n_elem, 0, sizeof(int)*n_buckets);
84 memset(global_starting_position, 0, sizeof(int)*n_buckets);
85 num_threads = n_buckets;
86 omp_set_num_threads(num_threads);
87 limit = dim;
88 w = limit/n_buckets;
89 A = (int *) malloc(sizeof(int)*dim);
90 B = (int *) malloc(sizeof(int)*dim);
91 for(i=0;i<dim;i++) {
92   A[i] = random() % limit;
93 }
94 if (dim < 10000000) {
95   printf("Unsorted data \n");
96   for(i=0;i<dim;i++) {
97     printf("%d ",A[i]);
98   }
99   printf("\n");
100 }
101 //local buckets, n_buckets for each thread
102 buckets = (struct bucket *) calloc(n_buckets*num_threads, sizeof(struct bucket));
103 // ****
104 // Starting the main algorithm
105 // ****
106 t1 = omp_get_wtime();
107 #pragma omp parallel
108 {
109   num_threads = omp_get_num_threads();
110   int j,k;
111   int local_index; // [0 : n_buckets)
112   int real_bucket_index; // [0 : n_buckets * num_threads)
113   int my_id = omp_get_thread_num();
114   workload = dim/num_threads;
115   int previous_index;
116   #pragma omp for private(i,local_index)
117   for (i=0; i< dim;i++){
118     local_index = A[i]/w;
119     if (local_index > n_buckets-1)
120       local_index = n_buckets-1;
121     real_bucket_index = local_index + my_id*n_buckets;
122     buckets[real_bucket_index].n_elem++;
123   }
124   int local_sum=0;
125   for (j=my_id; j< n_buckets*num_threads; j=j+num_threads){
126     local_sum += buckets[j].n_elem;
127   }
128   global_n_elem[my_id]=local_sum;
129   #pragma omp barrier
130   #pragma omp master
131   {
132     for (j=1; j<n_buckets; j++){
133       global_starting_position[j] = global_starting_position[j-1] + global_n_elem[j-1];
134       buckets[j].start = buckets[j-1].start + global_n_elem[j-1];
135       buckets[j].index = buckets[j-1].index + global_n_elem[j-1];
136     }
137   }
138   #pragma omp barrier
139   for (j=my_id+n_buckets; j< n_buckets*num_threads; j=j+num_threads){
140     int previous_index = j-n_buckets;
141       buckets[j].start = buckets[previous_index].start + buckets[previous_index].n_elem;
142       buckets[j].index = buckets[previous_index].index + buckets[previous_index].n_elem;
143   }
144   #pragma omp barrier
145   #pragma omp for private(i, b_index)
146   for (i=0; i< dim ;i++){
147     j = A[i]/w;
148     if (j > n_buckets -1)
149       j = n_buckets-1;
150     k = j + my_id*n_buckets;
151     b_index = buckets[k].index++;
152     B[b_index] = A[i];
153   }
154 #pragma omp for private(i)
155   for(i=0; i<n_buckets; i++)
156     qsort(B+global_starting_position[i], global_n_elem[i], sizeof(int), cmpfunc);
157 }
158 total = omp_get_wtime() - t1;
159 tmp = A;
160 A = B;
161 B = tmp;
162 if (dim < 1000000) {
163   printf("Sorted Data \n");
164   for(i=0;i<dim;i++) {
165     printf("%d ",A[i]);
166   }
167   printf("\n");
168 }
169 printf("Sorting %d elements took %f seconds\n", dim,total);
170 int sorted = 1;
171 for(i=0;i<dim-1;i++) {
172   if (A[i] > A[i+1])
173     sorted = 0;
174 }
175 if (!sorted)
176   printf("The data is not sorted!!!\n");
177 }

```

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.

8876	9877	98767	98769	98770	98771	98772	98773	98775	98776	98777	98778	98782	98783	98784	98785	98786	98787	98789	98790	98797	98798	98799	98800	98801	98801	98804	98810		
8881	9881	98816	98817	98817	98817	98823	98823	98825	98825	98825	98825	98827	98827	98829	98829	98830	98830	98831	98831	98832	98832	98833	98833	98834	98842	98842	98847	98848	
8884	9885	98850	98851	98851	98851	98852	98852	98853	98853	98853	98853	98855	98855	98857	98857	98862	98862	98864	98864	98865	98865	98868	98868	98869	98870	98873	98873	98875	98877
8888	98886	98886	98887	98887	98887	98888	98888	98889	98889	98889	98889	98891	98891	98893	98893	98897	98897	98898	98898	98900	98900	98904	98904	98905	98905	98907	98908	98911	98918
8892	98920	98920	98921	98921	98921	98922	98922	98923	98923	98923	98923	98924	98924	98925	98925	98927	98927	98928	98928	98929	98929	98930	98930	98931	98931	98932	98932	98935	98937
8956	98957	98958	98959	98960	98960	98962	98962	98963	98963	98963	98963	98964	98964	98965	98965	98967	98967	98968	98968	98969	98969	98970	98971	98972	98973	98975	98976		
8989	98991	98991	98993	98993	98994	98994	98996	98996	98996	98997	98997	98998	98998	99001	99002	99003	99003	99007	99007	99007	99007	99008	99008	99008	99008	99009	99009		
9191	99020	99020	99022	99022	99023	99027	99028	99032	99032	99033	99033	99037	99037	99040	99042	99043	99043	99044	99044	99045	99045	99055	99055	99056	99056	99057	99057	99058	99058
9056	99066	99067	99068	99068	99070	99070	99071	99071	99073	99073	99074	99074	99077	99077	99077	99077	99078	99078	99079	99079	99080	99080	99081	99081	99083	99083	99088	99089	
9997	99996	99996	99999	99999	99999	99999	99999	99999	99999	99999	99999	99999	99999	99999	99999	99999	99999	99999	99999	99999	99999	99999	99999	99999	99999	99999	99999		
9913	99133	99133	99133	99133	99133	99133	99133	99133	99133	99133	99133	99133	99133	99133	99133	99133	99133	99133	99133	99133	99133	99133	99133	99133	99133	99133	99133		
9915	99153	99153	99154	99154	99154	99154	99154	99154	99154	99154	99154	99155	99155	99155	99155	99156	99156	99156	99156	99156	99156	99156	99156	99156	99156	99156	99156		
9919	99191	99191	99192	99192	99192	99192	99192	99192	99192	99192	99192	99192	99192	99192	99192	99192	99192	99192	99192	99192	99192	99192	99192	99192	99192	99192	99192		
9921	99231	99231	99232	99232	99232	99232	99233	99233	99235	99235	99236	99236	99236	99236	99237	99237	99239	99239	99240	99242	99242	99242	99243	99245	99246	99247	99249		
9258	99250	99250	99260	99260	99261	99261	99262	99262	99265	99265	99267	99267	99267	99271	99271	99272	99274	99274	99276	99276	99276	99277	99277	99277	99277	99277	99277		
9986	99287	99287	99290	99290	99293	99293	99295	99295	99298	99298	99300	99301	99304	99305	99305	99310	99310	99311	99311	99311	99311	99311	99311	99311	99311	99311	99311		
9932	99322	99322	99323	99323	99323	99323	99325	99325	99326	99326	99326	99326	99327	99327	99329	99329	99329	99329	99329	99329	99329	99329	99329	99329	99329	99329	99329		
9942	99401	99401	99404	99404	99406	99406	99409	99409	99410	99410	99411	99411	99411	99411	99411	99411	99411	99411	99411	99411	99411	99411	99411	99411	99411	99411			
9433	99434	99434	99435	99435	99437	99437	99438	99438	99441	99441	99442	99442	99443	99443	99445	99445	99446	99446	99447	99447	99448	99448	99449	99449	99450	99450	99451	99451	
9473	99474	99475	99478	99478	99480	99481	99481	99483	99483	99483	99484	99484	99486	99486	99488	99488	99489	99489	99490	99490	99491	99491	99492	99492	99493	99493	99494	99494	
9504	99504	99505	99508	99508	99510	99511	99511	99512	99512	99513	99513	99514	99514	99515	99515	99515	99515	99516	99516	99517	99517	99518	99518	99519	99519	99520	99520		
9524	99527	99527	99530	99530	99531	99531	99532	99532	99533	99533	99537	99537	99538	99538	99539	99539	99540	99540	99541	99542	99542	99543	99543	99544	99544	99545	99545		
9508	99508	99508	99509	99509	99510	99510	99511	99511	99512	99512	99513	99513	99514	99514	99515	99515	99516	99516	99517	99517	99518	99518	99519	99519	99520	99520			
9669	99664	99664	99664	99664	99664	99664	99664	99664	99664	99664	99664	99664	99664	99664	99665	99665	99665	99665	99665	99665	99665	99665	99665	99665	99665	99665	99665		
9674	99675	99676	99676	99677	99677	99678	99678	99679	99679	99679	99680	99680	99681	99681	99682	99682	99683	99683	99684	99684	99685	99685	99686	99686	99687	99687			
9705	99705	99706	99706	99707	99707	99707	99708	99708	99709	99710	99711	99711	99712	99712	99713	99713	99714	99714	99715	99715	99716	99716	99717	99717	99718	99718			
9738	99738	99740	99741	99741	99744	99744	99747	99747	99749	99749	99758	99752	99754	99754	99755	99755	99757	99757	99758	99759	99759	99760	99761	99761	99762	99762	99763		
9741	99741	99741	99742	99742	99743	99743	99744	99744	99745	99745	99746	99746	99747	99747	99748	99748	99749	99749	99750	99750	99751	99751	99752	99752	99753	99753			
9817	99817	99817	99820	99820	99823	99823	99825	99825	99826	99826	99827	99827	99828	99828	99829	99829	99830	99830	99831	99831	99832	99832	99833	99833	99834	99834			
9856	99857	99858	99858	99860	99861	99861	99862	99862	99863	99863	99865	99865	99867	99867	99868	99868	99869	99869	99870	99870	99873	99873	99877	99877	99880	99880			
9887	99887	99887	99888	99888	99889	99889	99891	99891	99894	99894	99895	99895	99897	99897	99898	99898	99899	99899	99900	99900	99901	99901	99902	99902	99903	99903			
9911	99911	99911	99911	99911	99912	99912	99912	99912	99913	99913	99914	99914	99915	99915	99916	99916	99917	99917	99918	99918	99919	99919	99920	99920	99921	99921			
9956	99956	99957	99958	99958	99959	99959	99960	99960	99961	99961	99962	99962	99963	99963	99964	99964	99965	99965	99966	99966	99967	99967	99968	99968	99969	99969			
9993	99994	99994	99994	99994	99994	99994	99995	99995	99996	99996	99996	99996	99997	99997	99997	99997	99997	99997	99997	99997	99997	99997	99997	99997	99997	99997			

Sorting 18,000 elements took 0.01487 seconds

- **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.

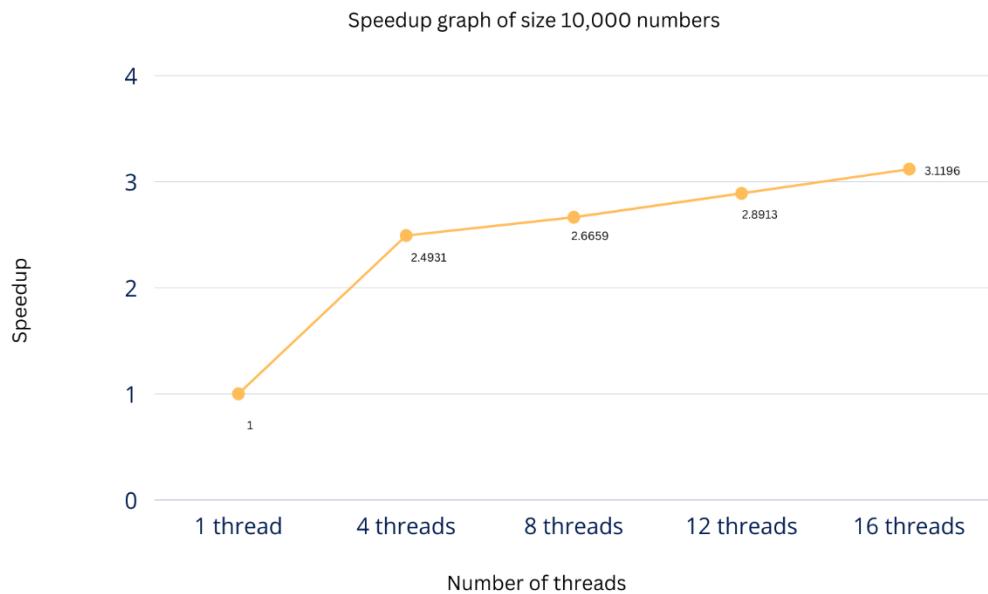
- **8 threads** with 0.024327 seconds of the execution time.

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

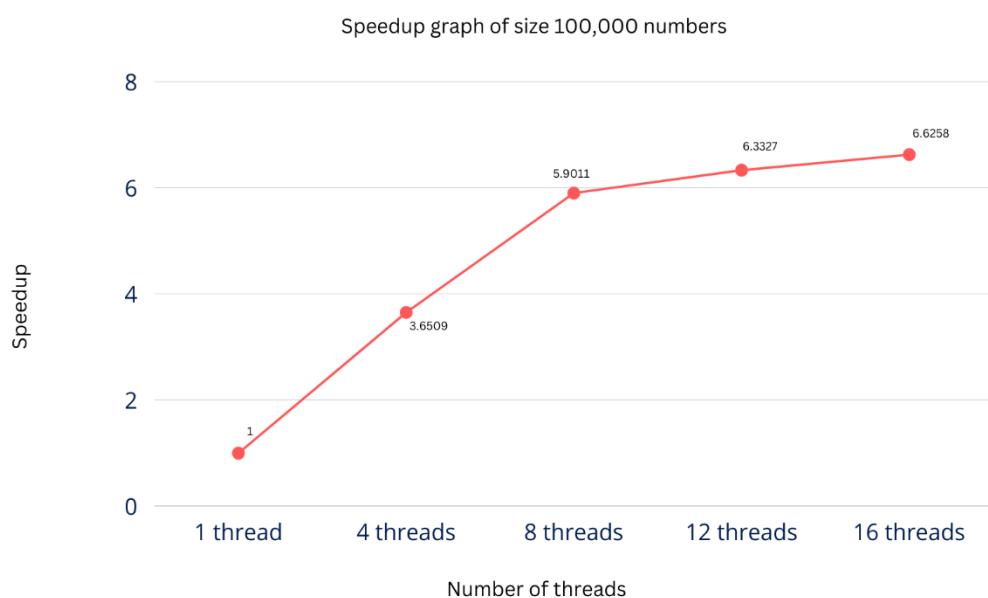
- **16 threads** with 0.075496 seconds of the execution time.

Speedup graph

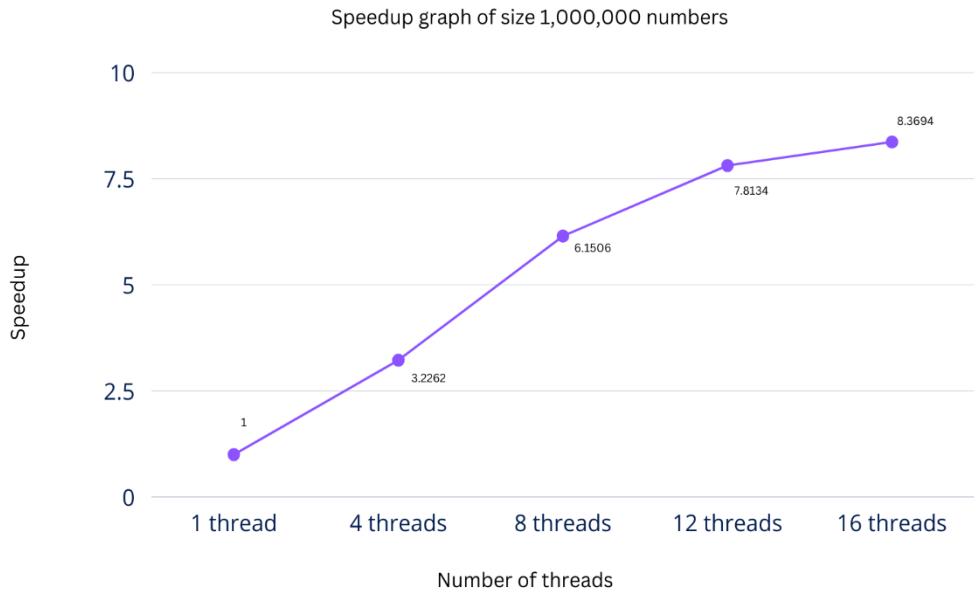
- Speedup graph of size 10,000 numbers



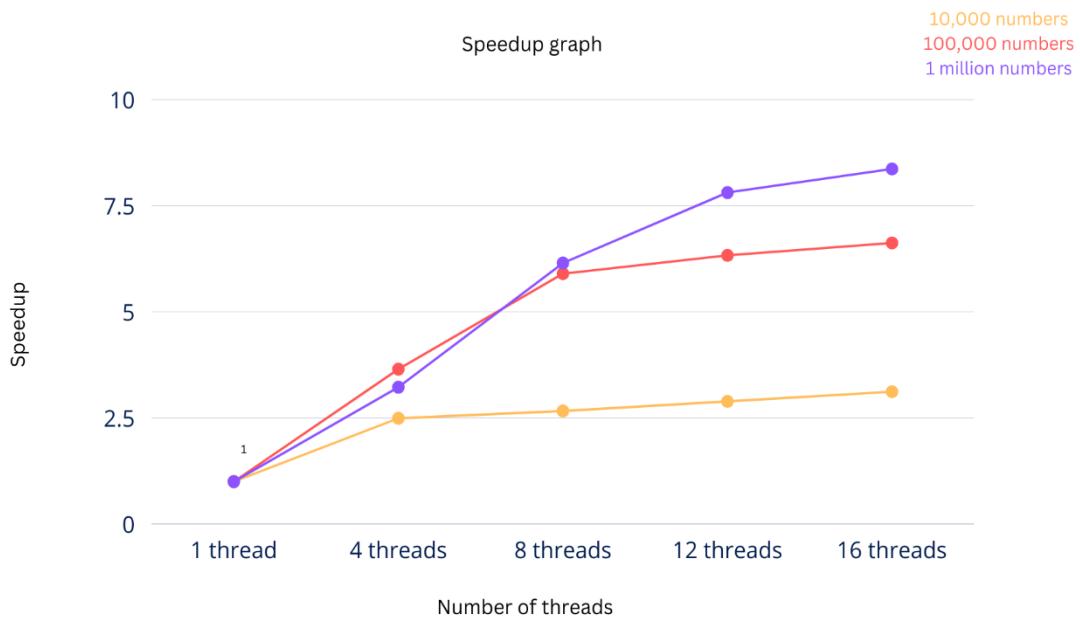
- Speedup graph of size 100,000 numbers



- 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.