Bucket Sort

Algorithm Description

In order to sort n random generated numbers in an ascending ordered manner by distributing them into certain amount of buckets. Assume the algorithm uses m buckets, with the ith bucket representing key values in the range [i k ... (i +1) k −1], for 0 ≤ i ≤ m − 1. Each bucket is sorted evenly and individually. After all the buckets have been sorted, concatenate all the buckets and check whether all the numbers are sorted in the right order. Last but not the least, the sorted list has to be checked whether it is in the right order, using for loop to iterate through the whole list in order to make sure the entire list is in an ascending order.

Description of Parallel Approach

There are three parts of the program that can decrease the running time by distributing the workload and using parallel computing. First, instead of letting one process generated all n random numbers, use n process generate its own portion of random numbers. Secondly, let the number of buckets equals to the number of processors and do a semi-bucket sort, which put the numbers into a sorted then conduct a sequential sort in each processor simultaneously. Thirdly, in the checking part, each process does its own checking and then communicate with each other.

Assume that n input elements are distributed across p processes, with each process initially holding n/p unsorted elements generated by each process itself. Each process now has multiple little buckets (one per process that it may receive data from): these little buckets were referenced as boxes. Set the number of boxes on each process to be m = p. Each process places its n/p key values into p boxes with key value x going to the box number $\lfloor x/(k/p) \rfloor$. Using all to all communication, for each process i, where $0 \leq i \leq p-1$, send the jth box to process j, where $0 \leq j \leq p-1$, $i \neq j$. After each process receives p-1 boxes from all other processes, combining them with its box and them sorts all of them together using sequential bucket sort to make one list of approximately n/p key values. Similar to the sequential algorithm, the sorted list has to be checked whether it is fully sorted as well, but in a parallel manner. For each processor, it should check its order in a sequential manner. In addition, it has to be checked that the last element of a process is larger than the first element of the next processor in order to make sure the entire array has all the element in order.

Experimental Setup

a. Describe the machine
   Node00 is a 24-core node on onyx cluster which is composed by two 12 core processors running at 1213.062 MHz on average with the maximum capability of 3100MHz, configured with 32 GB RAM. Each core has 2 level of cache, 64K for the 1 level cache (43K for L1d, 32K for L1i), 256K for 2nd level cache. All cores shared 15360K for the third level cache.

Each node installed installed 64-bit Intel Xeon quad-core CPU which would be able to run at 3.1-3.2GHz. Each of them is a 4-core node with 8GM RAM and 250GB Disk.

b. List experiments planned
   i. Time measurement of serial program running with number of iteration =20,000,000, 100,000,000, 200,000,000, 400,000,000, seed = 12345 and bucket number equals to 40,000. Request for 16 nodes, run the program on node20 with different number of processors (n=2, 4, 8, 16) as well as parallel and using MPI to utilize the parallel functionality.

Data
Execution time (s)

Table 1 Execution Time (s)

	n=20000000	n=100000000	n=200000000	n=500000000
N=1	1.88685	11.24381	23.934184	64.461521
N=2	1.551095	8.730703	18.893457	49.331533
N=4	1.259178	5.418464	11.275444	27.71161
N=8	1.70491	8.394272	17.608716	44.43981
N=16	0.37794	1.62657	3.29415	8.784859

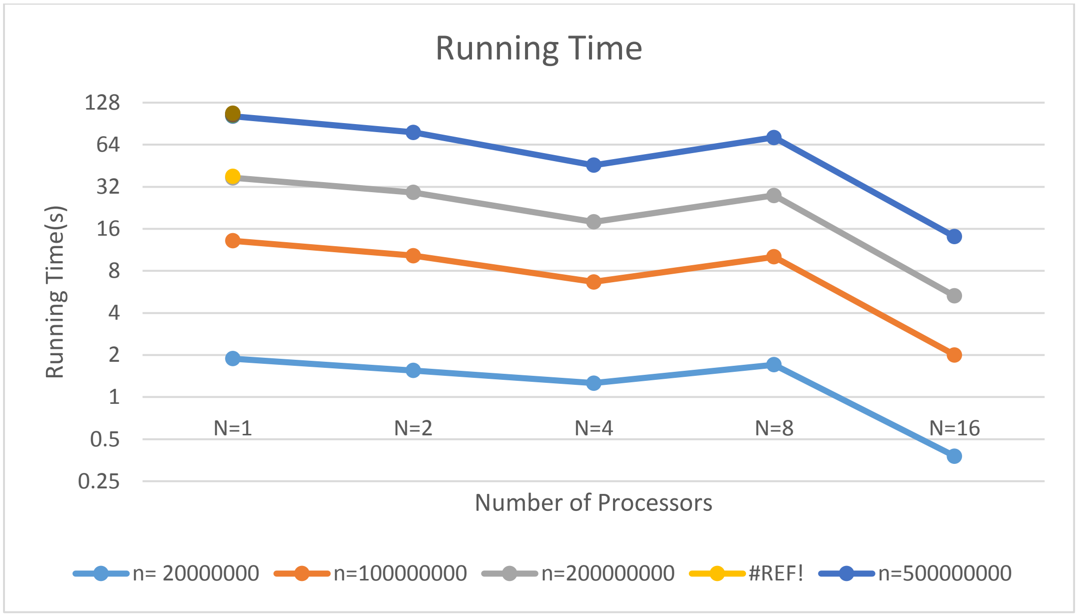

Figure 1. Executing Time vs Number of Processors

According to the Figure1, with larger number of processors, the running time goes down. However, it is not inversely proportional to the number of processors being used. The larger number of buckets it runs with should have shorter running time, however, it is not promised, since it can be seen that the running time goes up and down while the number of buckets keeps decreasing.

Speed Up

Table 2. Speed Up vs. Number of Processors

	n=20000000	n=100000000	n=200000000	n=500000000
N=1	1	1	1	1
N=2	1.216463	1.287847	1.266797495	1.306700138
N=4	1.498478	2.075092	2.122682176	2.326155752
N=8	1.106715	1.339462	1.359223694	1.450535477
N=16	4.992459	6.91259	7.265663069	7.337798023

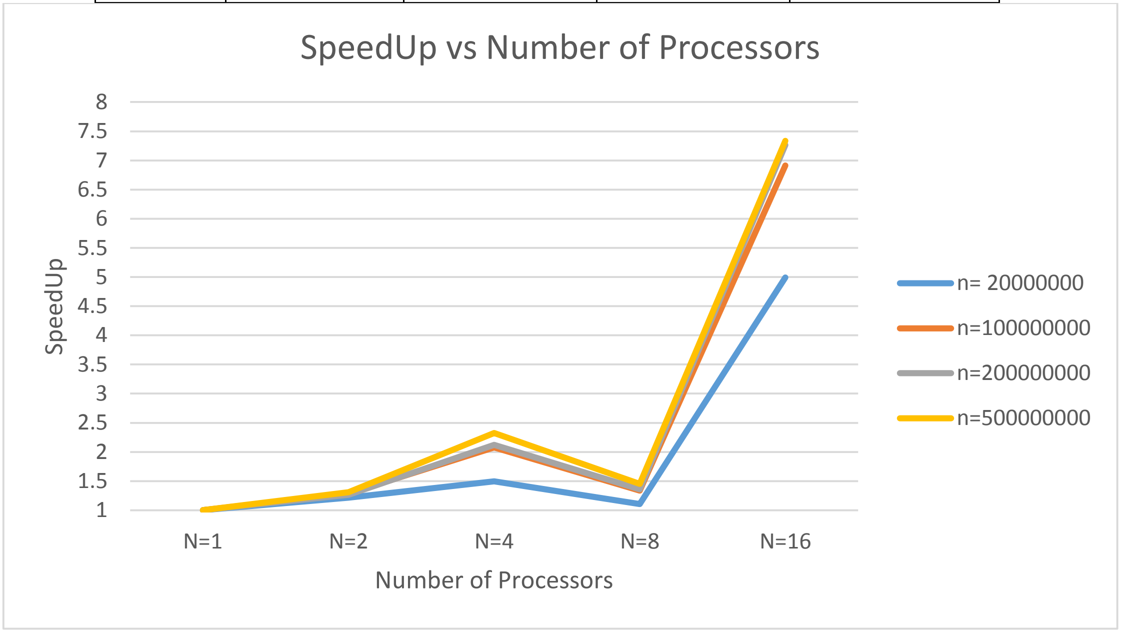

Figure 2. Speed Up vs. Number of Processors

The second figure above shows the relation between speed up and the number of processors being used in the program. We didn’t get a linear speed up, especially when processor number equals to 8, the curve went down. The larger number of processor used in the program, the better speed up tended to have. With the same amount of iterations, but different number of buckets, the speed up varies while the number of bucket varies. From ProcessNumber =8 to ProcessNumber = 16 the slot of the line is really steep which informed that the speed up improved significantly from 8 to 16.

Efficiency

Tale 3. Efficiency vs. Number of Processors

	n=20000000	n=100000000	n=200000000	n=500000000
N=1	1	1	1	1
N=2	0.608232	0.643924	0.633398748	0.653350069
N=4	0.374619	0.518773	0.530670544	0.581538938
N=8	0.138339	0.167433	0.169902962	0.181316935

N=16	0.312029	0.432037	0.454103942	0.458612376

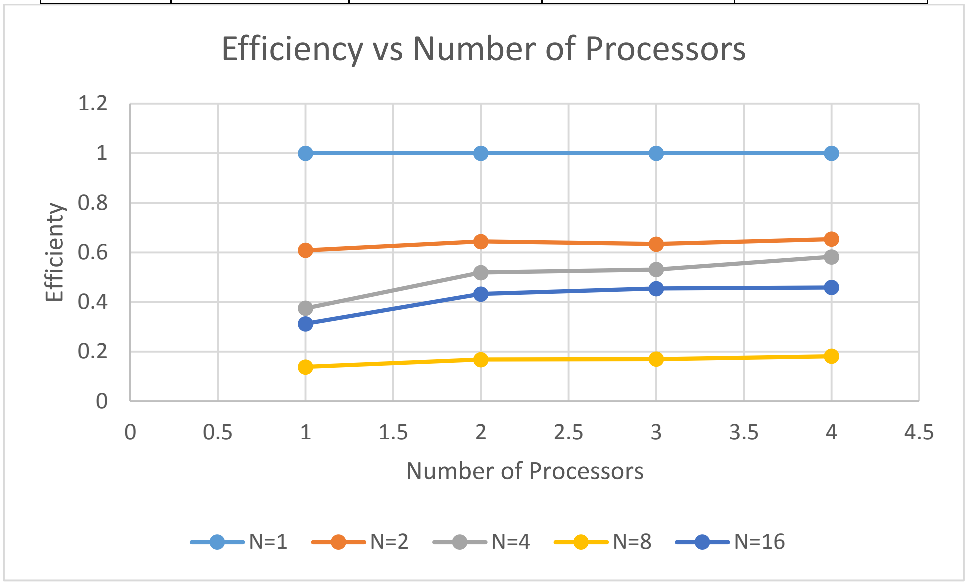

Figure 3. Efficiency vs. Number of Processors

It is shown in Figure 3 that when number of processors equals to 8, the efficiency is the lowest comparing to others. The highest efficiency of a parallel program occurs when 2 processors are utilized.

Conclusion

The more processors being used to run a program, the shorter running time it tends to get, as well as a better speed up. However, considering the cost of using more processors, more processors might not be the ideal plan to get the best efficiency. When running a program with certain amount of processors, other than speedup, efficiency should also be considered as a vital fact.