**Performance Comparison of Sorting Algorithms**

**Group Members**

Kanwar Muzammil 20k0469

Ahmed Abdullah 20K0470



**Teacher Name: Sir Nadeem Kafi Khan**

**Performance Comparison Of Sorting Algorithms**

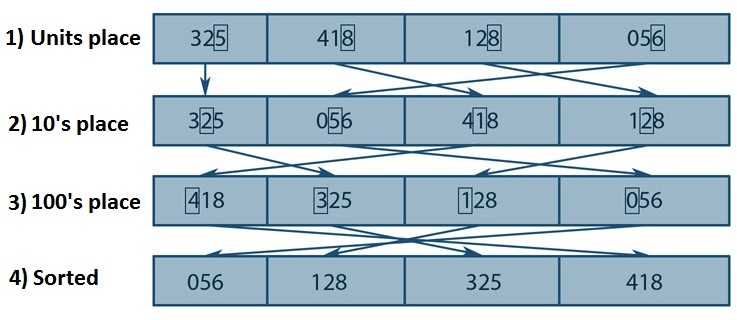
# Introduction

In this project, we implemented parallel programming on sorting random numbers using three different sorting algorithms and compared their performance. The performance of each algorithm is subjective of the time taken to sort the data. We tested quick sort, merge sort and radix sort algorithms. We changed the input size on all 3 sorting algorithms and displayed the time taken by changing the number of threads and processors. Time taken for both open mp and mpi is displayed after sorting the array

# RadixSort

The time taken for sorting by comparison based sorting algorithms such as merge sort and quick sort is in the order of **O(n log (n)).** The counting sort algorithm does a better job when the range of the n items is within n. However, when the range is increased to **n2**, the performance of counting sort is worse than the comparison based sorting algorithms.

Radix sort solves this problem by performing the counting sort iteratively for each digit from units place to kth place. Thus, the time taken by radix sort is in the order of **kn** instead of **n2**.

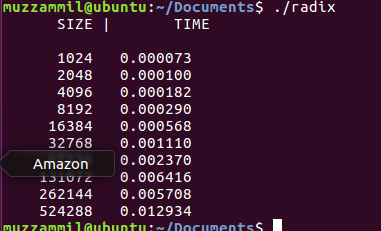


In this algorithm, first we find the maximum number in the entire range of numbers. If the maximum number has k digits, the iterations on each digit are performed from units place to kth place. In each iteration, we find the number of times each single digit from 0-9 is repeated in the units place of all numbers and save it as an array. The cumulative sum of the weights of the numbers 0-9 is calculated. At this point, we know the relative position of each single digit in the sorted output. However, the items with the same units place numbers still maintain the order from the initial array. The second iteration aligns the items based on 10's place and third digit iteration arranges with respect to 100's place. This process goes on until the kth place. In the end, we get a sorted series of numbers after k iterations. Since, there is no comparison between the numbers in this sorting algorithm, the time taken for sorting is dependent only on the number of elements and the number of digits in the maximum number.

**Radix sort (OpenMp)**

**Size=1 million**

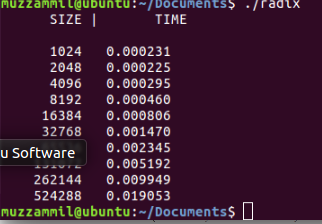
Threads=10



Threads=50

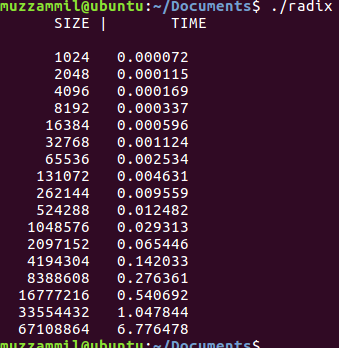


Threads=100

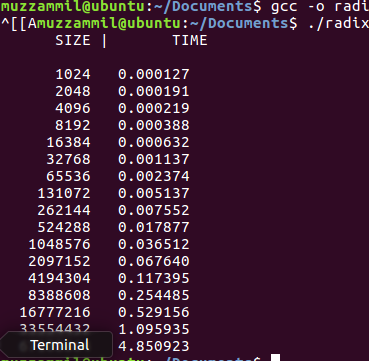


**Size=100 million**

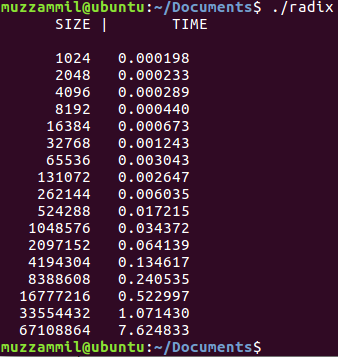
Threads=10



Threads=50

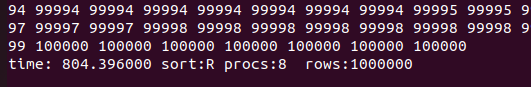


Threads=100

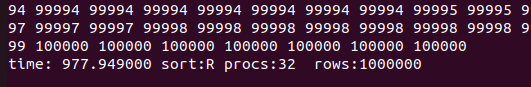


**Size=1 million**

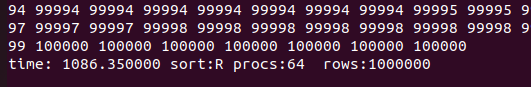
processes=8



processes=32

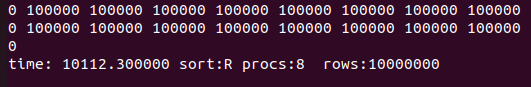


processes=64

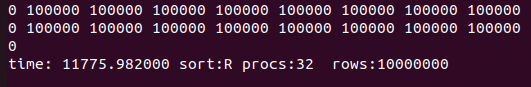


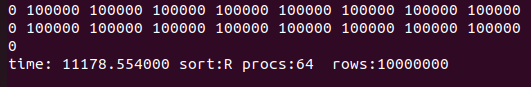
**Size=10 million**

processes=8



processes=32



processes=64

# MergeSort Algorithm

MergeSort algorithm is also a divide and conquer algorithm. The sorting algorithm divides the array of data into equal number of parts and sorts them. As part of dividing the data equally, it reaches to the point where there is only one number of data in an array. Since, there is only



3

43

27

38



82

9



27



10

10



38



3

43



82

9



27

27

38



43

3

3

43



82

9

82

9

10

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 38 | 27 | 43 | 3 | 9 | 82 | 10 |

one data in an array, it is sorted.

Now, second part of the algorithm is    to merge the sorted data. As the



38



10

data are merged, they are sorted as well while merging until all of the data is merged back into single array. The schematic of the algorithm is shown in the figure on the right.

|  |  |  |  |
| --- | --- | --- | --- |
| 3 | 27 | 38 | 43 |

|  |  |  |
| --- | --- | --- |
| 9 | 10 | 82 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 9 | 10 | 27 | 38 | 43 | 82 |

# Parallelization Method and Programming Flow:

We used OpenMPI as a Message Passing Interface(MPI) to parallelize the sorting algorithms.

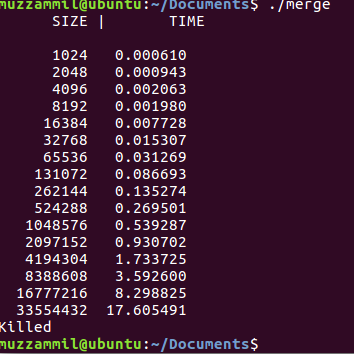
The initialization of the process takes place in the root node. Root node(id=0) takes the array size from command argument given by the user and creates an array using the rand () function:

Minimum number and maximum number decides the range of elements in the array.

**Merge sort (OpenMp)**

**Size=1 million**

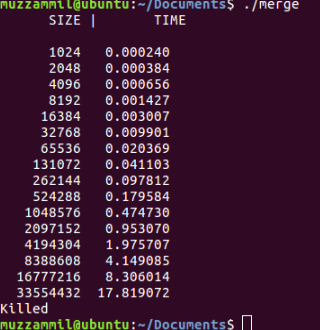
Threads=10



Threads=20

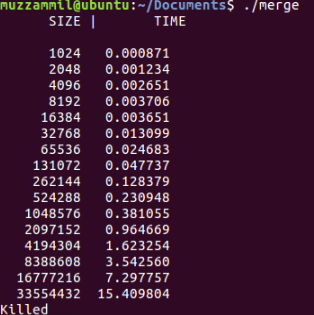


Threads=30

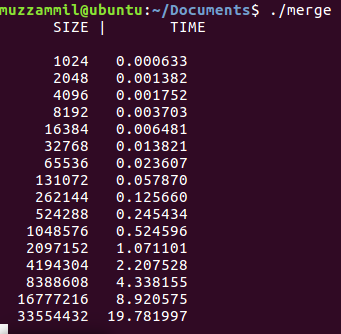


**Size=100 million**

Threads=10



Threads=20

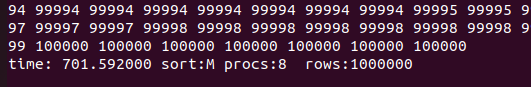


Threads=30

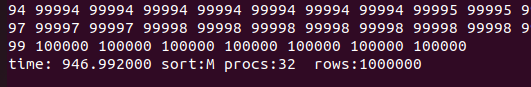


**Size=1 million**

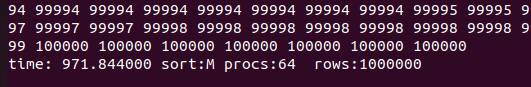
processes=8



processes=32

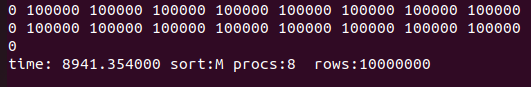


processes=64

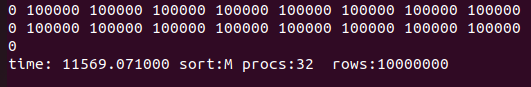


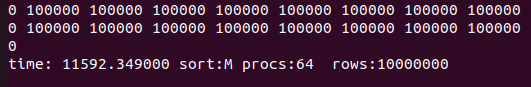
**Size=10 million**

processes=8

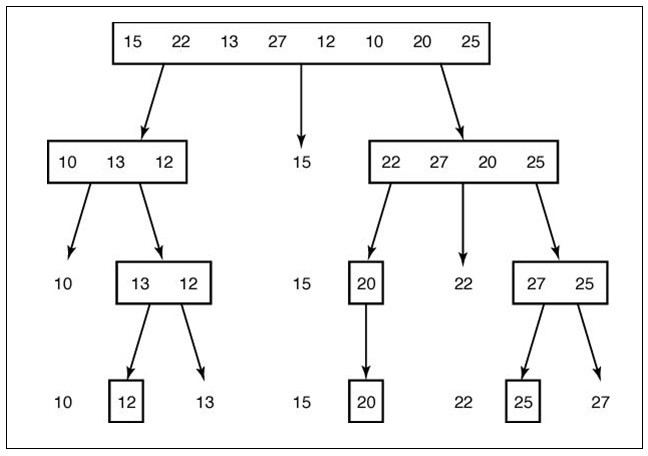


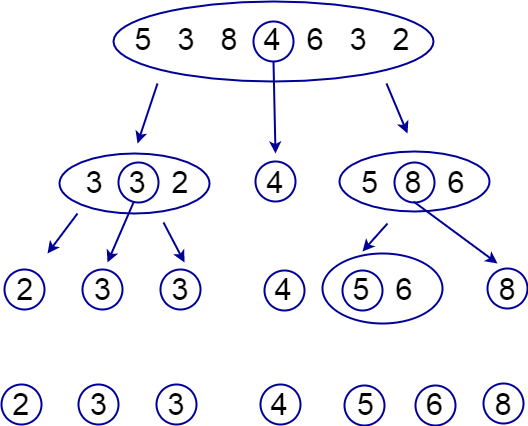
processes=32



processes=64

# Quicksort

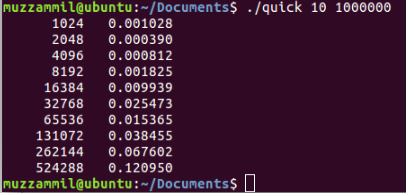
Quicksort divides the array into two partitions and then sorts each partition recursively. The array is partitioned by placing all items smaller than a pivot item before the pivot and all items larger than the pivot after the it. From every such partition process, the position of pivot gets fixed. The pivot can be any item, and for convenience we simply make it the first one.



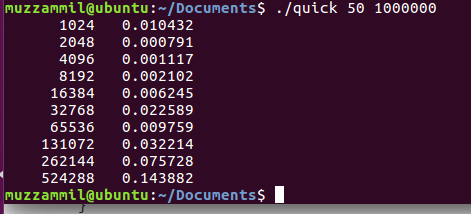
**Quick sort (OpenMp)**

**Size=1 million**

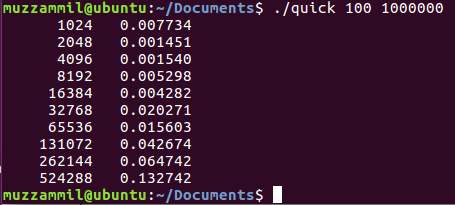
Threads=10



Threads=50

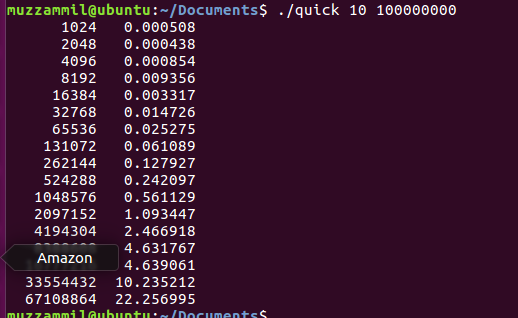


Threads=100

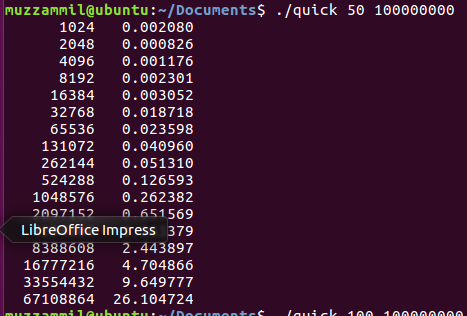


**Size=100 million**

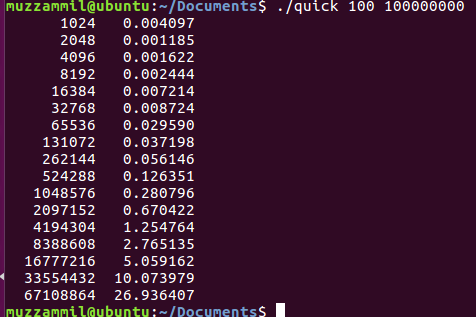
Threads=10



Threads=50

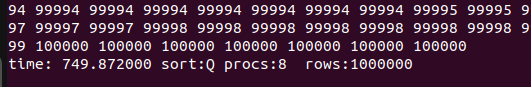


Threads=100

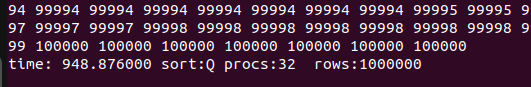


**Size=1 million**

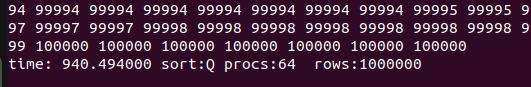
processes=8



processes=32

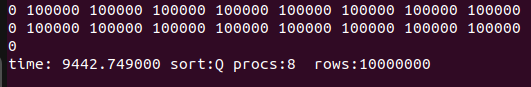


processes=64

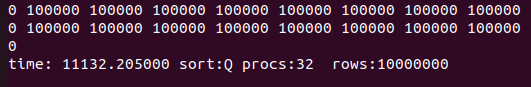


**Size=10 million**

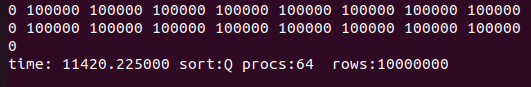
processes=8



processes=32



processes=64

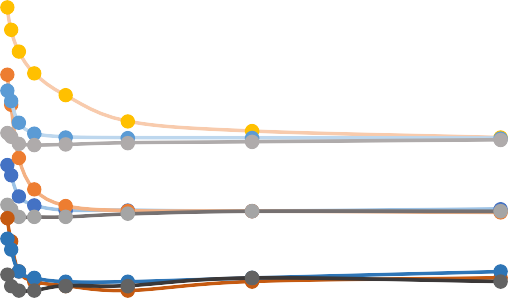


# *mpirun -n <number\_of\_nodes> <executive\_file\_name> <sorting\_technique{Q,M,R}>*

*<printing/not printing data{P,N}> <size\_of\_array>*

# Results and Discussion:

|  |
| --- |
|  |
|  |
| 10million data  points |
| 1million data  points |



Processing Time vs. Number of Processors

6

5

4

3

2

1

0

0

50

100

150

MergeSort\_10mil QuickSort\_10mil RadixSort\_10mil QuickSort\_1mil MergeSort\_1mil

RadixSort\_1mil

NumberofProcessors

Graph above shows that as the number of data points increases the calculation time increases as well. As the number of processor increased, the calculation time decreased. However, as the number of processors is more than 16, the calculation time remained constant for higher number of processors. This can be because the communication between the processors becomes the bottleneck.

Radix sort was the fastest algorithm among all three algorithms when it comes to less number of processor used. But, quicksort seemed to work better when there is greater number of processors available. When the number of processors increased to 128, the Radix sort performance decreased; however, the other two algorithms showed some improvement.



Processing Time vs. Data Size

800

700

600

500

400

300

200

100

0

-0

MergeSort QuickSort

Radix Sort

20000

40000

60000

80000 1000000

10000000

Data Size

Time(s)

The graph above shows the processing time for each algorithm using 32 processors for different data size. It shows that for upto 10 million data size, the quick sort performed better than other two algorithms. However, for the higher number of data, the Radix sort performed better.

When we increased the number of digits of our data, mergesort and quicksort performance was not affected as much. However, in Radix sort, when we sorted the six digit numbers, the performance of the algorithm decreased.

# Conclusion:

We have presented the performance of three sorting algorithms using parallel processing. From our analysis, quicksort and mergesort perform well for large number of data. Quicksort performance depends upon the initial order of the array of data whereas the Radix sort depends upon the number of digit of the highest number. With increase in number of processor, performance of quicksort is better than the mergesort algorithm. The performance of Redix algorithm increased upto 16 processors and then the performance decreased upon further increase in number of processors. Hence, we should choose the sorting algorithm based on the number of data to be processed and the number of processors available.

# References:

1. Basu, D. (2013). Parallel Radix Sort on the GPU using C AMP. Retrieved December 18, 2016, from https://[www.codeproject.com/articles/543451/parallel-radix-sort-on-the-gpu-](http://www.codeproject.com/articles/543451/parallel-radix-sort-on-the-gpu-) using-cplusplus-amp
2. Quick Sort Operation - C And C | Dream.In.Code. (n.d.). Retrieved December 18, 2016, from <http://www.dreamincode.net/forums/topic/45353-quick-sort-operation/>
3. **For radix sort implementation**
   * [radix.4up.pdf (princeton.edu)](https://www.cs.princeton.edu/courses/archive/spring02/cs226/lectures/radix.4up.pdf)
   * [Radix Sort — Data Structures and Algorithms in C++17 2.0 documentation (kurttest.com)](https://data-structs.kurttest.com/notes/radix-sort.html)
   * [Sorting algorithms/Radix sort - Rosetta Code](https://rosettacode.org/wiki/Sorting_algorithms/Radix_sort)
4. **For implanting OpenMP directives**
   * [OpenMP support - IBM Documentation](https://www.ibm.com/docs/en/xl-c-and-cpp-linux/13.1.3?topic=v1313-openmp-support)
   * [#pragma omp task - IBM Documentation](https://www.ibm.com/docs/en/zos/2.4.0?topic=processing-pragma-omp-task)
   * [Chapter 3: nowait (aalto.fi)](https://ppc.cs.aalto.fi/ch3/nowait/)