Performance Comparison Of Sorting Algorithms

Chaitanya Kolluru, Mohit Israni, Biswas Rijal

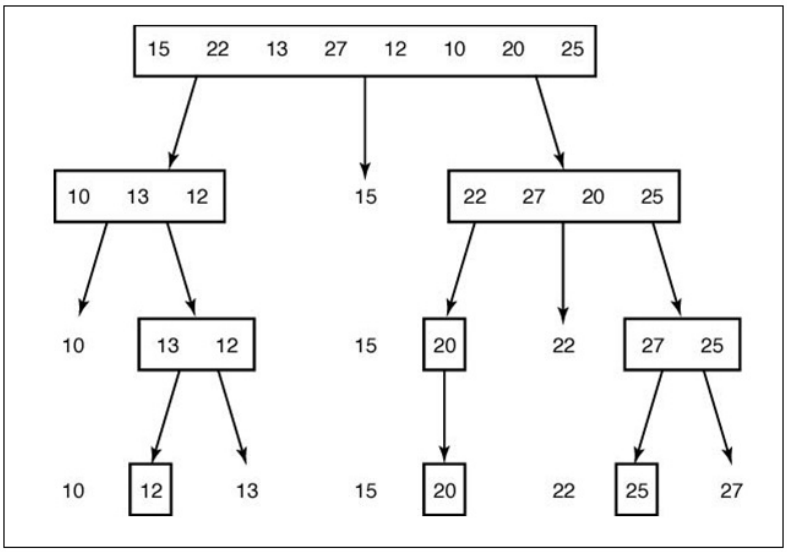
Introduction

The number of nanoscale transistors in a processor has almost reached it peak as it is very difficult to go to smaller scale transistors. In order to keep up with the increasing size of the data with the limited transistors per processor, it is essential to utilize the hardware efficiently. By combining large number of processors in to a supercluster, we can increase the available computing power by parallelizing and effectively distributing the work load on all the processors to achieve a goal. Even when quantum computing becomes accessible in the future, we need to use the parallelization techniques.

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. The three algorithms are as follows:

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.



To analyze the quicksort function, for a list of length ***n***, if the partition always occurs in the middle of the list, there will again be **log(*n*)** divisions. In order to find the split point, each of the ***n*** items needs to be checked against the pivot value. The result is ***n* log(*n*)**.

In addition, there is no need for additional memory as in the merge sort process.

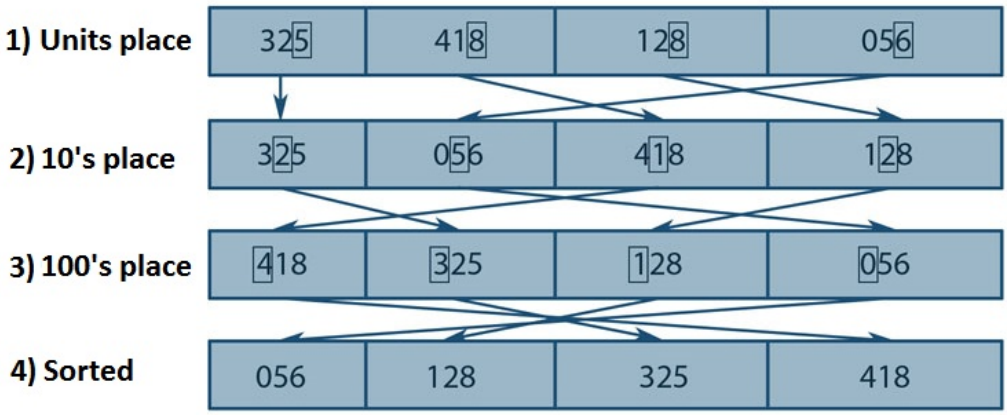
Unfortunately, in the worst case scenario, the split points may not be in the middle and can be very skewed to the left or the right, leaving a very uneven division. In this case, sorting a list of ***n*** items divides into sorting a list of **0** items and a list of ***n*−1** items. Then sorting a list of ***n*−1** divides

into a list of size **0** and a list of size ***n*–2**, and so on. The result is an ***O*(*n*^{2})** sort with all the overhead that recursion requires.

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 **n^2**, 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 **n^2**.



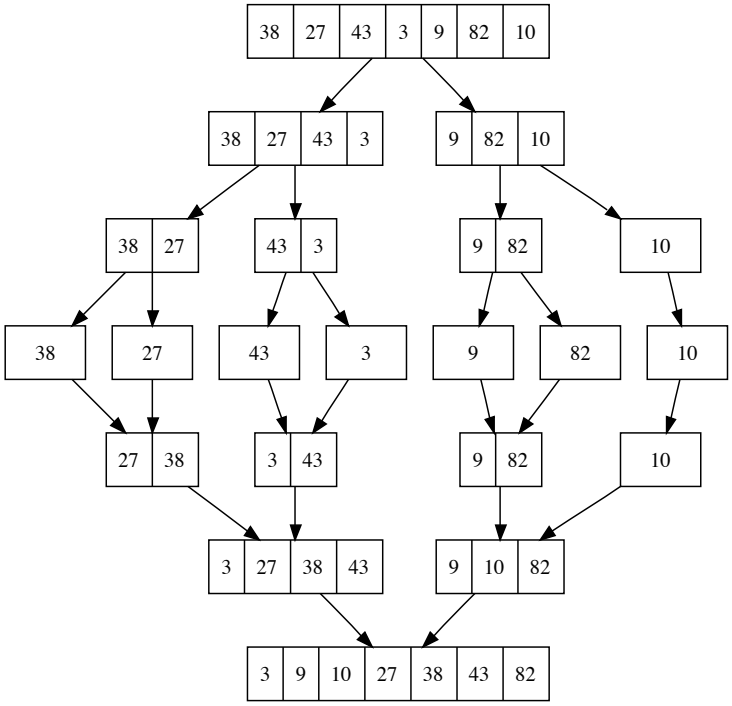
In this algorithm, first we find the maximum number in the entire range of numbers. If the maximum number has k digits, the iterations on each digit are performed from units place to kth place. In each iteration, we find the number of times each single digit from 0-9 is repeated in the units place of 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.

Performance Comparison Of Sorting Algorithms

Chaitanya Kolluru, Mohit Israni, Biswas Rijal

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 one data in an array, it is sorted. Now, second part of the algorithm is to merge the sorted data. As the 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.



Parallelization Method and Programming Flow:

We used OpenMPI as a Message Passing Interface(MPI) to parallelize the sorting algorithms. We parallelized the sorting algorithms by dividing the data equally into all the processors.

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 in C as below:

array= rand () % (max_num + 1 - min_num) + min_num;

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

The overall sorting process requires nodes of the format 2^n for maximum efficiency. Hence the nodes over 2^n remain are the non-active nodes. The root node divides and distributes the complete array equally amongst 2^n nodes.

Then all the nodes including the root node individually perform the sorting for their data set. Sorting technique used (quick, merge or radix sort) is specified by the user in the command line argument as:

mpirun -n <number_of_nodes> <executive_file_name> <sorting_technique{Q,M,R}> <printing/not printing data{P,N}> <size_of_array>

The string used to specify the sorting technique is compared using the function strcmp() as:

if (strcmp(<string_argument>, <string_sort>) == 0)

{sorting technique}

and the respective sorting technique is used in all the nodes. The strcmp() function requires including of <string.h> library.

Once all the arrays are individually sorted in their respective nodes, they are merged. Amongst active nodes, every even node sends its data to odd node where the merging of sorted arrays takes place. Merging takes place keeping the sorting order intact. The number of active nodes is now reduced to half. The process of sending and merging repeats for the remaining active nodes. At every merging step the number of active nodes reduces to half and that is why the number of nodes required for the overall sorting algorithm is required to be of form 2^n. The process continues till all the data is merged into the root node.

However, during merging there are lots of chances for one process to run faster than the other process which can lead to incorrect data message sending and receiving. Hence, all the data needs to be synchronously sent, which is done by using MPI_Ssend(). MPI_Ssend() will not return until matching receive is posted, or a handshake between send and receive takes place. Also, it likely gives best performance since this MPI implementation can completely avoid buffering data which is done when using MPI_Send().

Finally, the sorted arrays are printed by the root node. To calculate the overall time by the process, clocks were placed in the root node since the first step of creating array and final step of printing array in the overall sorting algorithm takes place in the root node. The difference in the end clock and start clock gives the time of the process.

Performance Comparison Of Sorting Algorithms

Chaitanya Kolluru, Mohit Israni, Biswas Rijal

Results and Discussion:

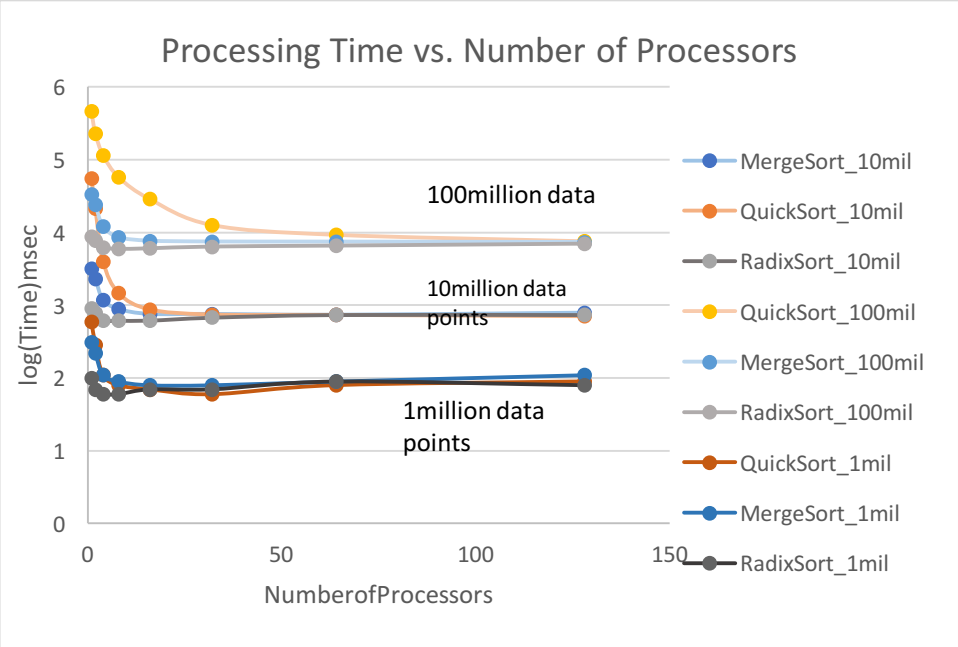


Figure 1 Time vs Number of processors

Figure 1. 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 asymptomatically. 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.

Figure 2 CPU time vs DataSize

The Figure 2 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.

We observed that the processing time differed for different runs for the same number of data. Thus, we took the median of the 3 runs and that what is presented in this report. We believe the reason behind it is that the processors that Hipergator allocates for the run were in different nodes which increased the data communication time in turn affecting our processing time.

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.

Performance Comparison Of Sorting Algorithms

Chaitanya Kolluru, Mohit Israni, Biswas Rijal

Supporting Documents:

1. C-programming sorting code *(Parallelized_sorting.c)*
2. Executable file *(algo.out)*
3. Raw Data in Excel *(Project_Data_HPC_Ver01)*

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-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/