Performance Analysis of Parallel Sorting Algorithms using MPI

Dr. Muhammad Hanif Durad[1], Muhammad Naveed Akhtar[2], Dr. Irfan-ul-Haq[3]
Department of Computer and Information Science (DCIS),
Pakistan Institute of Engineering & Applied Sciences (PIEAS)
[1]hanif@pieas.edu.pk, [2]naveed@ pieas.edu.pk, [3]irfanulhaq@ pieas.edu.pk

***Abstract*—Sorting is one of the classic problems of data processing and many practical applications require implementation of parallel sorting algorithms. Only a few algorithms have been implemented using MPI, in this paper a few additional parallel sorting algorithms have been implemented using MPI. A unified performance analysis of all these algorithms has been presented using two different architectures. On basis of experimental results obtained some guidelines has been suggested for the selection of proper algorithms.**

Keywords—parallel sorting; binary sort; bitonic sort; hyper quick sort; merge sort; odd even sort; regular sampling quick sort; radix sort; shell sort

I. INTRODUCTION

Sorting is one of the basic problems of computer science, and parallel algorithms for sorting have been studied since the start of parallel computing. Parallel sorting is one example of a parallel application for which the transition from a theoretical model to an efficient implementation is not straightforward. As a parallel application, the problem is especially interesting because it fundamentally requires communication as well as computation and is challenging because of the amount of communication it requires.

The possible methods of solving the sorting problem are broadly discussed here. The work by Knuth [1] gives a full survey of the data sorting algorithms. Among the latest editions we may recommend the work by Cormen et al.[2]. Given a wide range of parallel sorting algorithms and a large variety of parallel architectures, it is a difficult task to select the best algorithm for a particular machine and a specific problem instance. With the advent of multi-core machines the performance of MPI programs is very difficult to predict. However a statistical estimation may be used to foresee performance pattern of certain algorithms.

The main attention in this work is devoted to the study of parallel sorting methods including Bitonic Sort, Odd Even Sort, Shell Sort, Radix Sort, Hyper Quick Sort, Regular Sampling Quick Sort, Merge Sort, and Binary Sort. We limit our discussion to *internal sorting*, when all the ordered data on each processor may be fully located in main memory.

The rest of this paper is structured as follows: Section II summarizes related work, section III describes the target architectures and execution environment used for performance evaluation, while section IV reviews and compares the performance of individual parallel sorting algorithms for different number of processors. Section V presents unified analysis all algorithms based on experimental results. Section VI concludes the paper.

II. RELATED WORK

Parallel sorting algorithms have been studied extensively in the last three decades. Almost all textbooks on parallel computing discuss the sorting algorithms in detail. The classical references include [3], [4], [5], [6], [7]. Each of these texts usually presents a selected number of algorithms, thus getting a unified picture about the performance of these algorithms is bit difficult. A few other researchers have also discussed these topics, but they have highlighted different aspects of sorting.

NANCY M. AMATO et. al.[8] present a comparative performance evaluation of three different parallel sorting algorithms namely: bitonic sort, sample sort, and parallel radix sort. M.F. Ionescu and Klaus E. Schauser [9] have studied bitonic sort on modern parallel machines which are relatively coarse grained and consist of only a modest number of nodes, thus requiring the mapping of many data elements to each processor. Ezequiel Herruzo et. al. [10], suggest a new algorithm of arrangement in parallel, based on Odd-Even Mergesort, called division and concurrent mixes. The work has been carried out using 8 processors cluster under GNU/Linux.

Pasetto, Davide, and Albert Akhriev [11] provide a qualitative and quantitative analysis of the performance of parallel sorting algorithms on modern multi-core hardware. They consider several general-purpose methods with particular interest in sorting of database records and very large arrays, whose size far exceed L2/L3 cache.

In summary, the existing papers or texts have following limitations:

978-1-4799-7505-1/14 $31.00 © 2014 IEEE
DOI 10.1109/FIT.2014.46

- It is difficult to get a unified picture of the performance various parallel sorting algorithms using MPI.
- Only a few sorting algorithms have been implemented using MPI.
- Analysis of comparatively modest size vectors is present in literature.

In the nutshell, we believe that the paper will provide some extensions in theoretical background and practical implementation of parallel sorting algorithms.

III. Target Architectures and Execution Environment

The two machines namely a standalone machine and other computing cluster were used in experiments having the following architectures:

SGI Virtue: 2 x Intel Xeon Processor E5440 @ 2.83 GHz with 12 MB cache, 4 cores and 4 GB memory.

Computing Cluster:

Head Node: 2 x Intel Xeon Processors E5504 @ 2.00 GHz with 4 MB cache, 4 cores and 16 GB memory,

Cluster-Workers: 6 x Intel Core i5 Processors @ 2.67GHz with 8MB Cache, 4 cores and 4 GB memory each.

These algorithms were executed repeatedly on above systems using MPI. SGI VIRTUE system has 8 processing elements on a single board, while the computing cluster system has 32 processing elements. These processing elements communicate with each other over Gigabit Ethernet network. Datasets used were created using uniform random number generator and varied from 2^10 to 2^21 integer data elements. During execution processing elements were grouped as 1, 2, 4, 8, 16 and 32. Analysis is carried out in two halves first half till 8 processing elements for which Ethernet interface is not involved, both systems are part of this analysis. Second half of analysis is for more than 8 processing elements when network communication takes place.

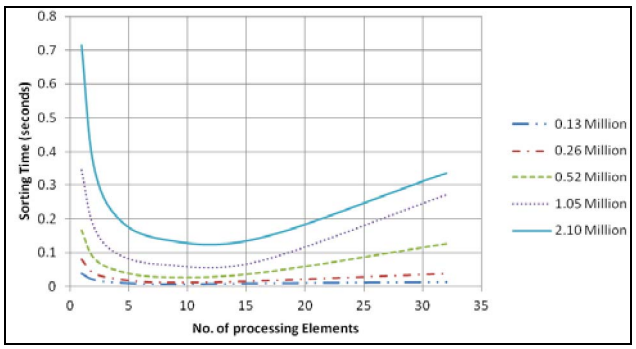

Fig. 1 Execution time for Bitonic Sort

IV. Review and Performance of Individual Parallel Sorting Algorithms

A large number of parallel sort algorithms are available in literature. The paper considers both stable and unstable sorting algorithms. For computational analysis in this section five different datasets varying from 0.13 to 2.1 million data items and only the cluster system has been used.

A. Bitonic Sort

A bitonic sequence [6] is a sequence of elements having property, that there exists an index *i*, such that all elements below this index are monotonically increasing and elements after this index are monotonically decreasing. Operation of splitting a bitonic sequence into two bitonic sequences is called a bitonic split. We can recursively perform bitonic split to get shorter bitonic sequences until we get a bitonic sequence of size 1. At this point output is sorted monotonically increasing. A total of log (n) operations are carried out for the purpose. This sorting algorithm is carried out in two phases.

1. In first phase an unsorted list is converted to a bitonic sequence using bitonic merging networks.
2. In second phase the bitonic sequence is sorted using bitonic merging networks

Fig. 1 shows the execution time for said data sets using Bitonic Sort. It depicts the usual MPI trend as the number of processing elements tends to reach 8, the execution time for Bitonic Sort increase due to network communication beyond this point.

B. Odd Even Sort

In Odd Even Sort [6], as 1st step processing elements sort their respective data independently using quick sort sequential algorithm, then processors exchange elements repeatedly in 2 phases. In 1st phase processors having odd ID (identity) performs compare exchange operation with the next neighboring processing elements which possesses even ID.

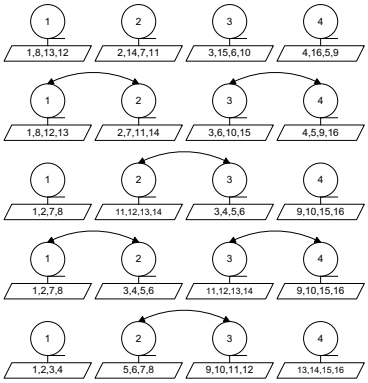

Fig. 2 Data sorting by means of Odd Even Sort method

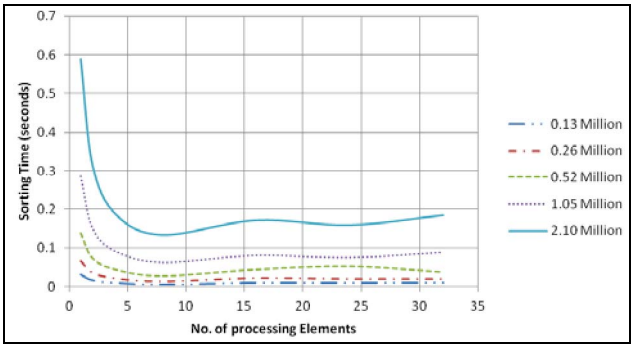

Fig. 3 Execution time for Odd Even Sort

In 2nd phase even numbered processors performs compare exchanges operation with the next neighboring odd processor. Both phases continue till the data is fully sorted. The whole process is elaborated in Fig. 2.

Fig. 3 shows the execution time for Odd Even Sort using same data sets. A usual MPI trend can be seen as the number of processing elements tends to reach 8. Increase in sorting time is beyond this point; however it has lesser slope than Bitonic and Shell sorts due less network communication between processes.

C. *Shell Sort*

A parallel variant of the Shell sort method is suggested in [6], using an N-dimensional hypercube. In this case sorting may be subdivided into two sequential stages. The interaction of the processors neighboring in the hypercube structure takes place at the first stage (N iterations). These processors may appear to be rather far from each other in case of linear enumeration. The required mapping the hypercube topology into the linear array structure may be implemented using the Gray code. The processors whose bit codes of their numbers differ only in position N-i are paired at each iteration i, $0 \leqslant i < N$. At the second stage the usual iterations of the parallel odd-even transposition algorithm are performed. The process is shown in Fig. 4.

The Fig. 5 shows the execution time for Shell Sort using same data sets. This figure depicts the usual MPI trend as the number of processing elements tends to reach

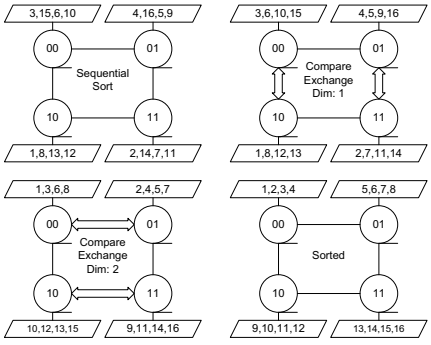

Fig. 4 Data sorting by means of Shell Sort method

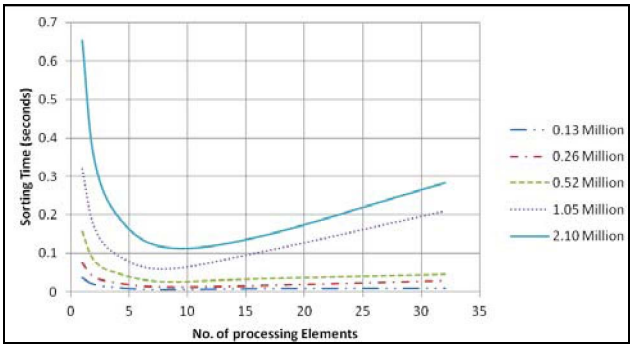

Fig. 5 Execution time for Shell Sort

8, the execution time increases due to network communication beyond this point. However a bit higher execution time is observed due to more communication overhead.

D. *Radix Sort*

This sorting algorithm is based on the binary representation of numbers to be sorted. During i^{th} iteration this algorithm sorts numbers according to i^{th} least significant bit. For radix sort to work properly it is require that the sorting method does not change the order of the input elements. In parallel formulation first the enumeration sort is performed using *prefix_sum()* and *parallel_sum()* procedures. Then repeatedly radix sort determines the position of elements with an r-bit value of j, by summing all the elements with same value and then assigning them to processes. Position of each element is remembered. At the end each processor sends its elements to the appropriate processor. Process labels determine the global sorting order.

The Fig. 6 shows the execution time for Radix Sort using same data sets. This figure shows a better trend of Radix Sort while the processing elements are below eight. After this point worse execution time is observed. This is due to *prefix_sum()* and *parallel_sum()* operations involved in sorting process over the network.

E. *Hyper Quick Sort*

Again hypercube architecture is used in this method as suggested in [7] . Let the initial data be distributed among

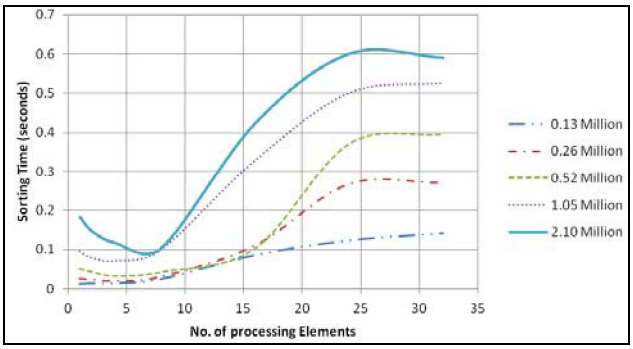

Fig. 6 Execution time for Radix Sort

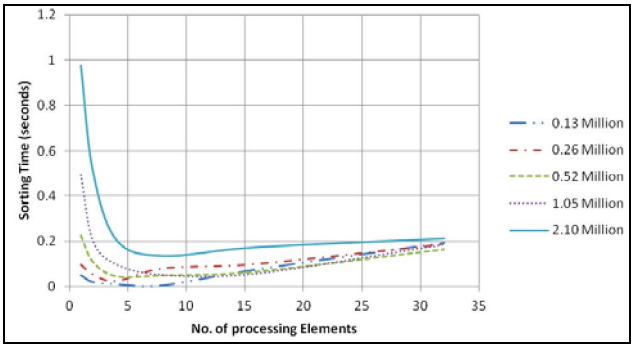

Fig. 7 Execution time for Hyper Quick Sort

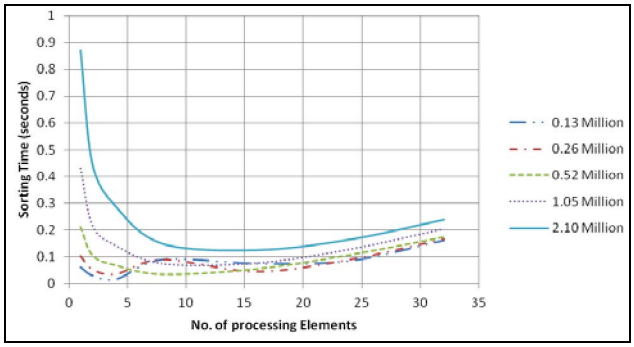

Fig. 8 Execution time for Regular Sampling Quick Sort

the processors in blocks of the size n/p. The resulting location of blocks corresponds to the enumeration of the hypercube processors. The first iteration of the parallel method is the following:

- Select the pivot element and broadcast it to all the processors (for instance, the arithmetic mean of the elements of some pivot processor may be chosen as the pivot element).
- Subdivide the data block available on each processor into two parts using the pivot element.
- Form the pairs of processors, for which the bit presentation of the numbers differs only in N position. After that the exchange of the data among these processors is executed.

As a result of executing this iteration, the initial data appear to be subdivided into two parts. One of them (with the values smaller than the pivot element value) is located on the processors, whose numbers hold 0 in the nth bit. There are only p/2 such processors. Thus, the initial N-dimensional hypercube also is subdivided into two sub hyper-cubes of N-1 dimension. The above described procedure is also applied to these sub hyper-cubes.

The Fig. 7 shows the execution time for Hyper Quick Sort using same data sets. Little peaks may be observed for smaller data sets due to pivot broadcasting.

F. Regular Sampling Quick Sort

The algorithm of the parallel sorting by regular sampling (PSRS) is also a generalization of the quick sort method [7] . These 4 stages are implemented to sort data with this variant of the quick sort algorithm:

1. Local data on the processors are sorted independently using quick sort algorithm. Then each processor forms a set of elements of its blocks with the indices 0, m, 2m,…,(p-1)m, where m=n/p(this set can be considered as regular samples);
2. All the data sets formed, are accumulated on a single processor and are sorted. Then the values of this set with the indices p+⌊p/2⌋-1,2p+⌊p/2⌋-1,… ,(p-1)p+⌊p/2⌋-1 form a new set of the pivot elements, and this set is transmitted to all the processors. At the end of the stage each processor partitions its block into p parts using the obtained set of the pivot values
3. Each processor sends the selected parts of its block to all the other processors.
4. Each processor merges the obtained parts in a single sorted block. After termination of this stage, the initial data become sorted.

The Fig. 8 shows the execution time for PSRS using same data sets. Initially both Hyper and Regular Sampling Quick sorts behave alike, however PSRS seems less efficient for higher number of processing elements.

G. Merge Sort:

Merge sort [5] uses divide and conquer approach for sorting. In this type of sort data is divided in 2 halves and assigned to processors this continues until individual numbers are obtained. After this each 2 pairs numbers are merged into sorted list each having 2 numbers. This sorted list is again merged to make 4 sorted numbers. This continues till the fully sorted 1 list is obtained. This algorithm perfectly maps to a tree structure shown in Fig. 9.

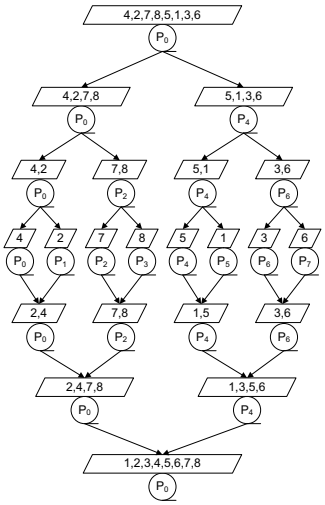

Fig. 9 Data sorting by means of Merge Sort method

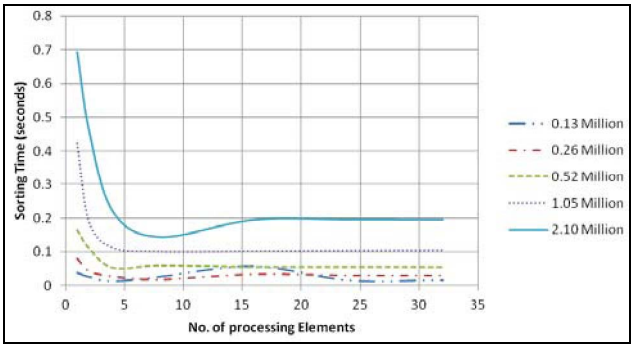

Fig. 10 Execution time for Merge Sort

The Fig. 10 shows the execution time for Merge Sort using same data sets. This figure represents an overall decreasing trend in execution time for merge sort as the number of processing elements increases. The slope increases when the processing elements start communicating over Ethernet, but still there is a decreasing trend for all data sets.

H. Binary Sort:

In this sorting mechanism all the data distribution and collection is same as that of merge sort algorithm. The only difference is that leaves in this sorting mechanism also performs sequential binary sort, while in merge sort quick sort is performed by the leaf nodes.

The Fig. 11 shows the execution time for Binary Sort using same data sets. This figure shows same trend as the Parallel Merge Sort because data distribution and collections procedures are similar for both methods. However it takes higher time than merge sort because leaves nodes also performs binary sort here.

V. UNIFIED PERFORMANCE ANALYSIS

A unified performance analysis of various parameters for all parallel sorting algorithms has been performed and the results are presented in this section:

A. Execution time versus number of processing elements (Cluster System)

A fixed data size of 2.1 million integer data items was

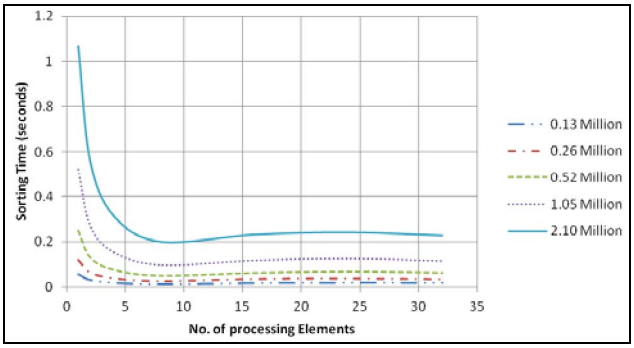

Fig. 11 Execution time for Regular Sampling Binary Sort

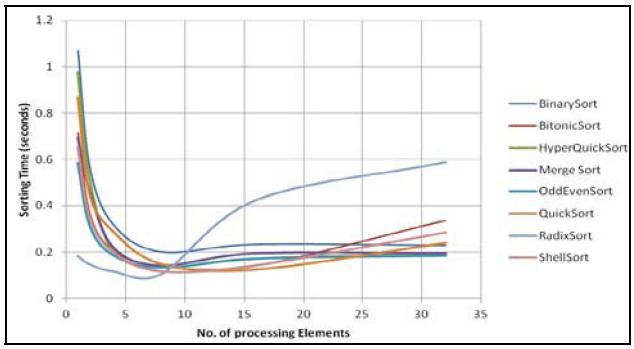

Fig. 12 Execution time vs no of processing elements

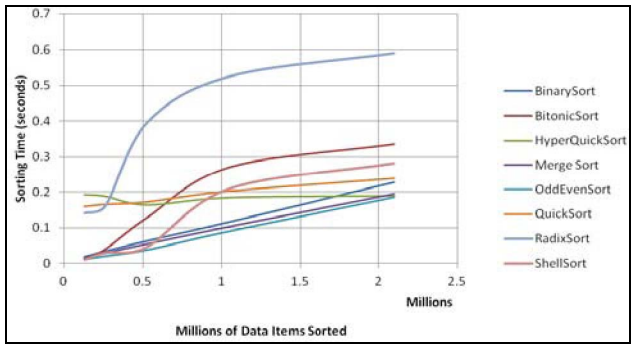

Fig. 13 Execution time versus number of data items

sorted using all algorithms for 1,2,4,8,16 and 32 processing elemets groups to observe execution time, the results are shown in Fig. 12.

The analysis results show that Radix Sort performs better than all other algorithms if there is no Ethernet traffic involved but it performs poor while processors starts communicating over Ethernet and Merge performs better even the network communication is involved.

B. Sorting time versus number of data items (Cluster System)

Different data sets were used to analyze the performance of these algorithms by using a group of eight processing elements. The results are given in Fig. 13.

The analysis results show that radix sort performs better for larger sets than all other sorting methods, while binary sort performs worst in this case due to leaves nodes performing sequential binary sort.

C. Timing analysis on Virtue System

A fixed data size of 2.1 million integer data items was sorted using all algorithms except Binary and Bitonic sorts (not implementable due to memory limitations) on SGI Virtue. The analysis was performed for a group of 1, 2, 4 and 8 processing elements to observe execution time and to verify out implementation on cluster system, the results are shown in Fig. 14 and Fig. 15.

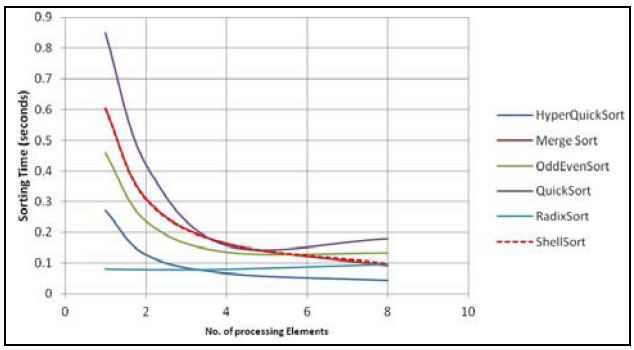

Fig. 14 Execution time versus no of processing elements

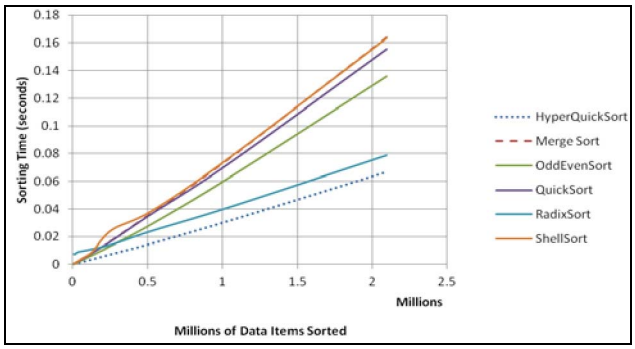

Fig. 15 Execution time versus number of data items

A close observation of graphs shows no commendable changes in performance but the overall reduction in execution time due to better processing elements.

VI. SUMMARY

In this paper we have analyzed most of the parallel sorting algorithms using MPI for two different architectures. It is very difficult to find a single paper implementing all these sorting methods altogether. Our findings from computational experiments performed in this regard are listed as:

1. There is a general decrease in execution time with increase in no of processing elements, and a general increase in execution time with increase in data size is observed if there is no network communication involved for all the sorting methods.
2. Radix sort performs better if there is no network communication and otherwise its performance is significantly affected.
3. A sudden increase in the execution time of Odd even sort is observed for eight processing elements. This is due to the saturation of local bandwidth used in MPI communication.
4. Performance of merge sort is scalable in both networked and local communication.

We had tried to get overall picture of most of the available parallel sorting algorithm, but we intent to extent our work by considering the following possible additions.

1. We are more interested to map experimental results to that of rigorous mathematical formulations of these algorithms.
2. We are ambitious to extend our work for high end state of the art machine architectures including GPUs.
3. A more in depth analysis needs to be carried out to compare hyper quick sort and regular sampling quick sort by varying the number of processing elements.
4. We are also interested to use better profiling tools in near future.

In short, we consider that this paper has contributed to a few extensions in practical implementation of parallel sorting algorithms.

REFERENCES

[1] Knuth, D. E., The Art of Computer Programming. Vol 3: Sorting and Searching, 2nd ed., Addison-Wesley, 1997.

[2] Cormen et al., Introduction to Algorithms, 2nd ed., MIT Press, 2001.

[3] Foster I., Designing and Building Parallel Programs, Addison-Wesley, 1995.

[4] Pacheco, Peter S. Parallel programming with MPI. Morgan Kaufmann, 1997.

[5] Wilkinson B., Michael A., Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers , 2nd ed., Prentice Hall, 2003.

[6] Grama A., Gupta A., Karypis G., and Kumar V., Introduction to Parallel Computing, 2nd ed., Addison-Wesley, 2003.

[7] Quinn, M. J., Parallel Programming in C with MPI and OpenMP, McGraw-Hill, 2004.

[8] Amato, Nancy M., et al. A comparison of parallel sorting algorithms on different architectures. Technical Report TR98-029, Department of Computer Science, Texas A&M University, 1996..

[9] onescu, Mihai F., and Klaus E. Schauser. "Optimizing parallel bitonic sort." Parallel Processing Symposium, 1997. Proceedings., 11th International. IEEE, 1997.

[10] Ezequiel Herruzo et. al.,A Message Passing Implementation of a New Parallel Arrangement Algorithm, World Academy of Science, Engineering and Technology,Vol:2 2008-09-29.

[11] Pasetto, Davide, and Albert Akhriev. "A comparative study of parallel sort algorithms." Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion. ACM, 2011.