Comparative Analysis of Sorting Methods using OpenMP

Edgar A. Guerrero, Abel Palafox, Juan P. Serrano, José L. Alonzo.
Centro de Investigación en Matemáticas, CIMAT A.C., Guanajuato, Gto., México.
E-mail: {guae, abel.palafox, jpsr, pepe}@cimat.mx.

Abstract

The development of new technologies has reduced the computing power limits. This allows handling large volumes of data. There are plenty of applications involving data sorting implementations. It is necessary to have suitable methods for efficiently sorting large data sets. Several sorting methods using computers with multiple processors using OpenMP technology are implemented. A comparative analysis of the computational cost of the sorting algorithms is presented. The results provide information to choose a sorting method suitable for the type of application and available computing power. We mention applications that require efficient sorting methods for large data volumes.

Keywords: Bubble sort, Odd-Even Transposition Sort, Rank Sort, Counting Sort, Bitonic Sort, Quicksort, Radix Sort, Merge Sort, OpenMP.

1. Introduction

Computation power has increased widely in the span of about 20 years and still grows. There are many areas requiring computational speed including numerical simulation of scientific and engineering problems. Problems such as weather forecasting have a specific deadline for the computations. Modelings of large DNA structures and quantum mechanic simulations have grand challenge problems due to the high amount of data. Additionally, analysis by finite element method, Monte Carlo simulations, roughly molecular dynamics, among others, it requires computing expensive calculations on large data sets in order to give confident results. Computations must be completed within a suitable time. In manufacturing realm and engineering, calculations and other simulations must be achieved within minutes or even seconds.

Sorting procedure is an important component of many applications. Then there is a wide variety of sorting methods. On the other hand, parallel algorithms have been studied extensively in the last four decades [1-7]. Therefore, many of the sorting methods have a parallel version. However the parallelization itself is not enough to reach optimal results. We consider essential to develop a benchmark of sorting methods for choosing a suitable one based on the specific application and the available computer power. In this paper, eight sorting algorithms (Bubble Sort, Odd-Even Transposition Sort, Rank Sort, Counting Sort, Bitonic Sort, Quicksort, Radix Sort and Mergesort) are implemented in parallel using OpenMP platform. For comparison purposes the performance of the parallel implementation is compared to the serial implementation. Their performance for various array sizes is measured with respect to sorting time.

In this context, there are developments which analyze the performance of the sorting algorithms using other platforms GPU (Graphic Processing Unit) architectures, OpenCL (Open Computing Language) platform [15, 16, 17]. GPU and OpenCL devices provide better results than traditional mechanisms. For the scope of this work, analyzing sorting methods with those platforms is lead as future work.

This paper is structured as follows. In section 2 we briefly explain eight classical sorting methods. The results of our implementations are presented on the section 3. Conclusions and future work are discussed on section 4.

2. Methods

Brief descriptions of several sorting methods on serial and parallel version are presented. A comparison of execution times with different number of processors and different data sets are presented.

In this work the following sorting methods are studied:

1. Bubble Sort
2. Odd-Even Transposition Sort
3. Rank Sort
4. Counting Sort
5. Bitonic Sort
6. Quicksort
7. Radix Sort
8. Merge Sort

We denote as X an array of integers whose size is n. The number of processor is denoted by p.

2.1 Bubble sort

The bubble sort method compares each element of the array with its neighbor. This procedure moves the highest order elements towards the last positions of the list on each iteration. All the elements are compared until the array is sorted.

This method is expensive because the whole array is analyzed on each iteration, even already sorted elements. Bubble sort method is widely used due to its simplicity.

Placing an element on the sorted array takes n comparisons on average where n is the number of elements. Then, the complexity level is O (n^2).

The parallel version of the Bubble Sort algorithm executes p phases. On each phase, the element xi is compared with its neighbor xi+1 and i is moved every p places. As we can see, the array is divided into sub-arrays of size p and the serial Bubble Sort is executed on each sub-array. Then, the complexity level is $O(n^2/p)$. This parallelization idea leads to the Odd-Even Transposition Sort method.

2.2 Odd-Even Transposition Sort

The Odd-Even Transposition Sort is a variant version of the Bubble Sort. This method operates in two phases: odd and even. In the even phase, each element xi of the list X, where i is even, is compared with its right neighbor xi+1. In the odd phase the right neighbor xj+1 of xj is compared where j is odd.

The serial version of the Odd-Even Transposition Sort takes no advantage with regard to Bubble Sort. In the worst case, Odd-Even method has a complexity level of $O(n^2)$.

The parallel version is more efficient. Using p processors, the complexity level is about $O(n^2/p)$. So, using n processors, the complexity decreases to O(n).

An important aspect about Odd-Even is the fact that reduces concurrency. During the execution of each phase (Odd or Even), every process only access to the element xi and its neighbor xi+1. This is provided both phases do not run simultaneously.

The parallel version of Odd-Even method uses a flag variable. This flag indicates whether the list is sorted (i.e. whether there are swaps). The shared variable flag takes only a value 0 or 1. However, this increases the execution time due to concurrencies. The directive reduction of OpenMP lets several processors access to the flag variable. An increment on flag and the directive reduction are used instead assign the value 1.

2.3 Rank Sort

The Rank Sort is designed for arrays of unique elements. For every element xi on the input array, the algorithm calculates the total number of elements xj that are less or equal to xi. The computed number is called the rank of xi. To calculate the rank of a number, we need to compare it with every element on the array. Then, each element on the input is placed into a new array. The position of the element on the new array corresponds to its rank.

For sorting an array of size n, the algorithm uses n iterations of the principal loop. In every iteration the algorithm requires n comparisons. With this analysis, we say that the complexity of the method is $O(n^2)$. We note that the execution time does not depend on the initial configuration of the array.

The parallel version of the algorithm could be obtained by computing the rank of every element on a different processor. The algorithm uses two arrays (unsorted and sorted), which are on a shared memory location. Both arrays are available for all the processes at the same time.

In the parallel version the complexity level is about $O(n^2/p)$. The main loop is divided between p processors and then n comparisons are placed for calculating the rank for every element. Using n processors, the complexity level is O(n).

Nevertheless Rank Sort algorithm is designed for arrays of unique elements, it is possible to improve it in order to deal with arrays with non-unique elements. This modification could lead concurrencies. For instance the final array is analyzed in order to treat repeated elements.

2.4 Counting Sort

This algorithm is designed for sorting arrays of integers with values in range [1,m]. This method builds the histogram C for the array X. Histogram C is updated by computing a prefix sum along the histogram. Finally, the method computes the position of each element on a sorted array B using the C array.

Counting sort method requires an array C of size m to store the histogram. Building the histogram, as well as computing positions, requires only one access to each element of the array. On the other hand, the prefix sum is performed over the elements of C, so there are m computations. Then, the complexity order is O(n + m).

The parallel version of Counting sort methods requires a deeper work. For instance, the copy of array X to array B is done using p processors.

2.5 Bitonic Sort

Bitonic Sort is based on building bitonic sequences. A bitonic sequence has two subsequences, one has increasing order and the other one decreasing order. The bitonic method for a sequence of n=2k elements consists of three blocks. The first two blocks sorts the two halves of the sequence in ascending and, respectively, descending order. The third part merges the two sorted halves to obtain the sorted array.

The parallelization is done in both parts: on creating the blocks and merging them. The serial version of Bitonic Sort has a complexity level of O(n log2n) whereas parallel version has a complexity level of $O(n \log_2 n/p)$.

2.6 Quick sort.

The Quicksort method is based on the divide-and-conquer paradigm. This method first divides a large array into two smaller sub-arrays: low elements and high elements. The complexity level of this algorithm is O(n log n) on average, assuming there are just distinct elements in the array. The worst-case performance is $O(n^2)$.

There are three-step divide-and-conquer stages for sorting a typical sub-array $[x_i, ..., x_k]$ of a array X:

- Divide: Partition (rearrange) the array $[x_i, ..., x_k]$ into two (possibly empty) sub-arrays $[x_l, ..., x_{j-1}]$ and $[x_{j+1}, ..., x_m]$, such that each element of $[x_l, ..., x_{j-1}]$ is less than or equal to x_j. In the same way, x_j is less than or equal to each element of $[x_{j+1,} ... , x_m]$.
- Conquer: Sort the two sub-arrays $[x_l, ..., x_{j-1}]$ and $[x_{j+1}, ..., x_m]$ using recursive calls to Quicksort.
- Combine: Since the sub-arrays are sorted in place, no work is needed to combine them. The entire array X is now sorted.

An important aspect is how the partition is done. Choosing a random pivot index j does not require additional computation. Nevertheless this could increase the complexity level up to $O(n^2)$ for certain cases. The qsort function of the C language performs a partition which sorts the subarray in place.

For the parallel procedure, we divide the array of size n into p sub-arrays. Then we use the qsort procedure to sort every sub-array on a different processor at the same time. Finally, all sub-arrays are merged. The complexity level for the parallel version is O(n log n / p).

2.7 Radix Sort.

Radix Sort can be used to sort items that are identified by unique keys. This method works as follows:

- Take the least significant digit (or group of bits) of each key.
- Sort the array of elements based on that digit, but keep the order of elements with the same digit (this is the definition of a stable sort).
- Repeat the sort with more significant digits each time.

Its efficiency is O(k n) for n numbers with k or fewer digits.

2.8 Merge sort

The Merge sort method splits the array into many sub-arrays until just two elements are contained. Then the algorithm starts merging such sub-arrays while sorts them. The algorithm can be described in two parts: splitting the initial array and merging it on the sorted array. This algorithm has a complexity level O(n log n).

For the parallel version, the initial array is divided into p sub-arrays. Each processor sorts a sub-array using the serial Merge sort version. Once every sub-array is sorted, those are merging on a single array. So, the parallel version of this method has a complexity level of O (n log n/p).

Results and Discussion

The test cases are arrays of integers randomly chosen on the interval [1, 1000]. The size of the test arrays are powers of 2 on the interval [26, 217]. For every size, we execute all the methods presented in this work using 2, 4, 8, 16 processors and executions of the serial version. Our implementations were made on C and we use OpenMP for executing the parallel versions. All the test cases were run on the host described below.

Time and String Metrics		**Constant Metrics**	
Machine Type	x86_64	CPU Count	16 CPUs
Operating System	Linux	CPU Speed	2395 MHz

Operating System Release	3.2.0-24-generic	Memory Total	32938000 KB
		Swap Space Total	1020 KB

In order to obtain statistical information about the running time for each method we run 30 independent tests for each array size and for each number of processors. On each test, the same unsorted array was used to measure the execution times. With this information, we report mean value, standard deviation, maximum value and minimum value for each method.

Figure 1 presents the mean times obtained varying the array size in serial version. Figure 2 shows the elapsed times for each method on parallel version using two processors. For visualization purposes the results obtained from 4^{th} to 8^{th} methods are presented in figure 3. Similarly, the results obtained for 4, 8, and 16 processors are presented in figures 4 to 9 respectively.

The first three methods has the expected behavior since these algorithms are the order $O(n^2)$. However, Odd-Even and Rank Sort takes advantage of the parallelization and improves the execution time when increasing the number of processors. Indeed, execution time seems to be linear when increasing the number of processors. However this improvement is not enough.

The best results are obtained by Counting Sort, Bitonic Sort, Quicksort, Radix Sort and Merge Sort. The lowest execution time is reported by Counting Sort method. Low execution time of the Counting Sort method is due to the range of the numbers which are between 1 and 1000. Thus, the cost for the histogram computation is negligible. In future we suggest using a range of values comparable with the size of the array.

Figures 10 and 11 present the mean elapsed times for a test case of size 2^{17} varying the number of processors. For visualization purpose the results are presented figure 10 shows all methods and figure 11 shows from 4^{th} to 8^{th} methods. The information on figures 10 and 11 is important due to the following: it is expected that the parallelization of a procedure reduces the computational cost. However, as can be seen on figure 10 and 11 the parallelization could increase the execution time. For instance the Merge sort seems to have a not surprising performance. However, this method still takes advantage on the parallelization. This is, increasing the number of processors it is possible to reduce the execution time. On the other hand, the Quicksort algorithm has a good performance but it does not improve from 8 to 16 processors.

The Radix Sort has a execution time between the Counting and Bitonic Sort. This method is a good option for sorting based on the obtained results. In contrast figure 11 shows that the mean time gets worse even using just 2 processors.

Statistic information for the execution times on the case 2^{17} is computed. With this information robustness of every method is compared. Methods with lowest standard deviation (SD) are less sensitive to the sort on input data. By comparing the SD and mean values when varying the number of processors, robustness of every method according to the parallelization is obtained.

Table 1 shows the statistical results for the serial version. Table 2 presents the results for 8 processors and for 16 processors are presented in Table 3. Counting Sort, Bitonic Sort, Quicksort, Radix Sort and Mergesort have low values for the SD. Also, those methods are executed in short times. Counting Sort, Bitonic Sort, Quicksort and Mergesort algorithms keep their values for SD in all cases. The Radix Sort increments its initial SD value. On the other hand, the Bubble Sort, Odd-Even and Rank Sort algorithms have big values in all the statistical values.

4. Conclusions and Future Work

We implemented eight classical sorting algorithms. We focused on studying their performance on large data sets using parallel computing. Also, we analyzed the sensitivity of those algorithms when increasing the number of processors. The results obtained include statistical information on a specific case.

The sensitivity analysis and the statistic information provide additional elements on the comparisons. For instance, the Merge sort seems loose performance. However we can deduce that this method could offer better results using a higher number of processors. On the other hand, the Quicksort algorithm presents, in general, the best performance. Nevertheless, increasing the number of processors will not improve the results.

As future work, would be implemented the algorithms on MPI and CUDA technology which allow us to increase the size of the test arrays. Additionally, there are some areas which present particular structures along large data sets. The performance of sorting methods could be improved by taking advantage of such structures.

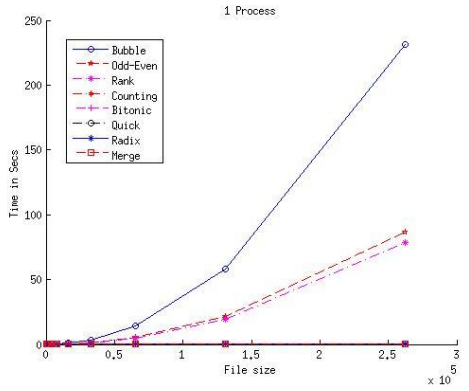

Figure 1. Elapsed times for serial versions.

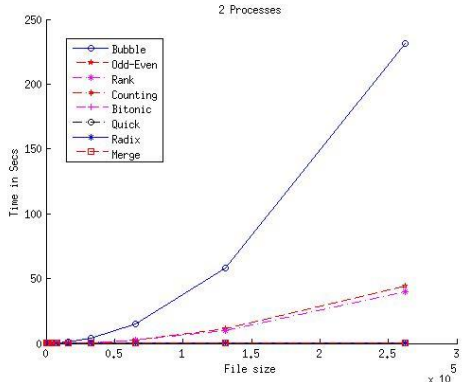

Figure 2. Elapsed times for two processors.

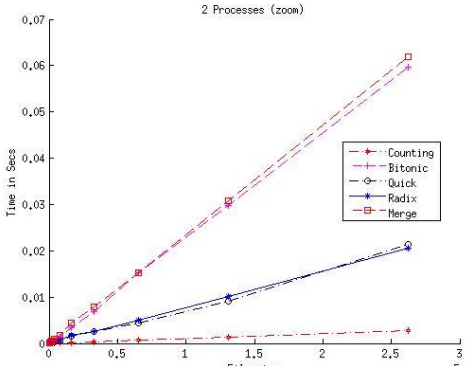

Figure 3. Elapsed time for two processors from 4th to 8th methods.

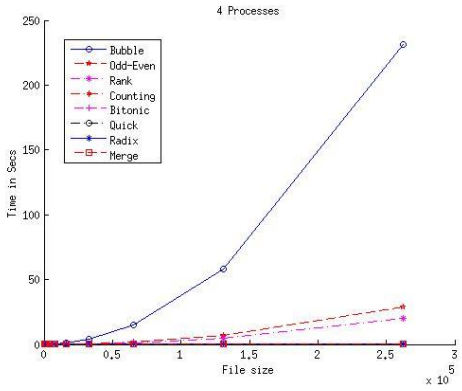

Figure 4. Elapsed times for four processors.

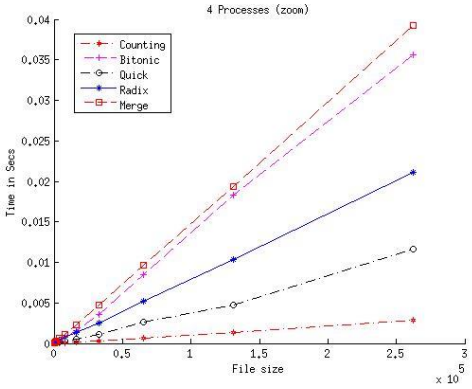

Figure 5. Elapsed time for four processors from 4th to 8th methods.

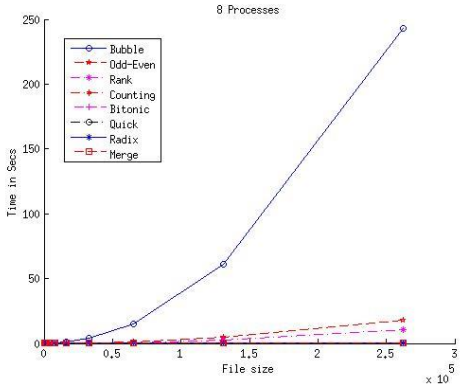

Figure 6. Elapsed times for eight processors.

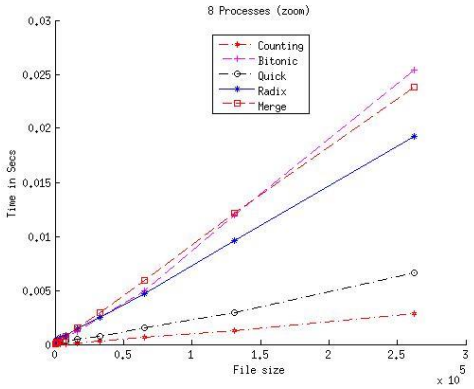

Figure 7. Elapsed times for eight processors from 4th to 8th methods.

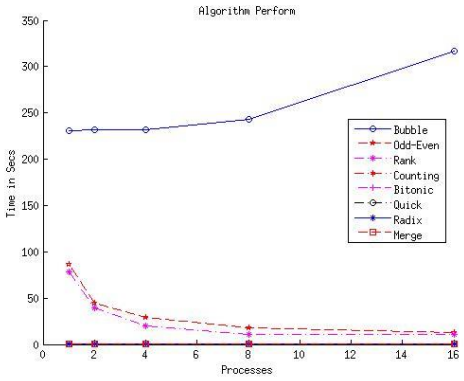

Figure 10. Algorithm performance varying processors number.

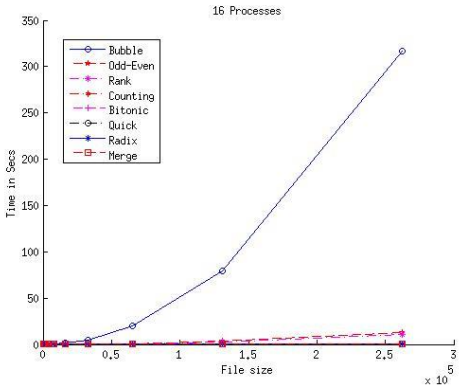

Figure 8. Elapsed times for sixteen processors

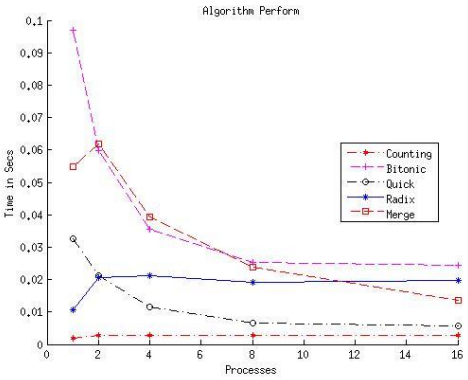

Figure 11. Algorithm performance varying processors number from 4th to 8th methods.

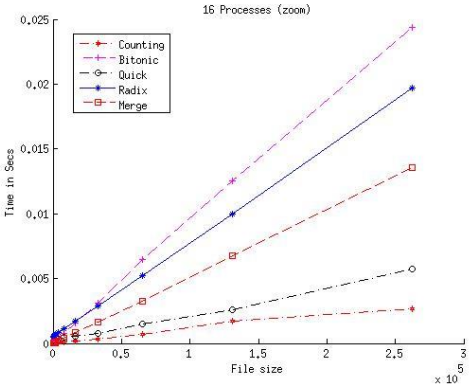

Figure 9. Elapsed times for sixteen processors from 4th to 8th methods.

Table 1 Stats for serial version on case 2^{17}.

Method	Min	Max	Mean	SD
Bubble	230.45	232.53	231.1843	0.60486
Odd-Even	86.035	86.872	86.3991	0.22909
Rank	78.497	79.189	78.6325	0.2239
Counting	0.001861	0.002044	0.0018792	3.5795e-05
Bitonic	0.095856	0.1116	0.096794	0.0028029
Quicksort	0.032015	0.04822	0.032622	0.0029462
Radix	0.010525	0.011296	0.010682	0.00015238
Merge	0.054602	0.055496	0.054921	0.00022916

Table 2 Stats for parallel version on case 2^{17} with 8 processors.

Method	Min	Max	Mean	SD
Bubble	242.92	243.71	243.2953	0.2165
Odd-Even	18.16	18.983	18.2453	0.1416
Rank	10.332	10.465	10.3478	0.025671
Counting	0.002506	0.0035	0.0028552	0.00028808
Bitonic	0.022366	0.030199	0.025362	0.0021564
Quicksort	0.005486	0.011502	0.0066391	0.0011887
Radix	0.018316	0.021488	0.019289	0.00057744
Merge	0.023099	0.024108	0.023795	0.00035635

Table 3 Stats for parallel version on case 2^{17} with 16 processors.

Method	Min	Max	Mean	SD
Bubble	316.17	318.39	317.2497	0.49886
Odd-Even	12.381	12.798	12.5655	0.10452
Rank	10.268	10.35	10.2772	0.020761
Counting	0.002608	0.003006	0.0026888	9.5532e-05
Bitonic	0.023041	0.029239	0.024378	0.0014214
Quicksort	0.004882	0.011883	0.0057337	0.0012452
Radix	0.018195	0.02245	0.019681	0.0010663
Merge	0.013186	0.013898	0.013541	0.00024272

References

[1] K. W. Batcher, "Sorting networks and their applications", *AFIPS Conf.* 32, (1968), pp. 307-314.

[2] H. Barlow, D. J. Evans and J. Shanehchi, "A parallel merging algorithm", *Information Processing Letters* 13, No. 3, Dec. 1981, pp. 103-106.

[3] G. Baudet and D. Stevenson, "Optimal sorting algorithms for parallel computers", *IEEE Trans. Comput*. C-27, No. I, Jan. 1978, pp. 84-87.

[4] F. Gavril, "Merging with parallel processors", *Comm. ACM* 18; 10, Oct. 1975, pp. 588-591.

[5] D. E. Muller and F. P. Preparata, "Bounds to complexities of networks for sorting and for switching", *J. Assoc. Comput. Math*. 22; 2, April 1975, pp. 195-201.

[6] L. G. Valiant, "Parallelism in comparison problems", *SIAM J. Comput*. 4; 3, Sep. 1975, pp. 348-355.

[7] Yasuura, N. Takagi and S. Yajima, "The parallel enumeration sorting scheme for VLS1", *IEEE Trans. Comput*. C-3, No. 12, Dec. 1982, pp. 1192-1201.

[8] B. Chapman, G. Jost, R. van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*, The MIT Press, (2008).

[9] B. Wilkinson, M. Allen, *Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers*, 2nd edition. Pearson Education Inc. (2005).

[10] B. Barney. OpenMP. (2012) https://computing.llnl.gov/tutorials/openMP.

[11] C. Breshears. *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*, O'Reilly Media, Inc., (2009).

[12] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson, *Introduction to Algorithms* (2nd ed.), McGraw-Hill Higher Education. (2001)

[13] Shi-Kuo Chang, *Data Structures and Algorithms*, World Scientific (2003).

[14] Karl Fürlinger, "OpenMP Application Profiling - State of the Art and Directions for the Future", *In Proceedings of the 2010 International Conference on Computational Science (ICCS 2010)*, Amsterdam, NL, May (2010).

[15] Khan F.G., O.U. Khan, B. Montrucchio, P. Giaccone, "Analysis of Fast Parallel Sorting Algorithms for GPU Architectures", *Frontiers of Information Technology (FIT)*, pp. 173 – 178. (2011).

[16] Y. Quan, D. Zhihui, S. Zhang, "Optimized GPU Sorting Algorithms on Special Distributions", *In 11th International Symposium on Distributed Computing and Applications to Bussines, Enginering & Science*, pp. 57-61, (2012)

[17] K. Zhang, J. Li, G. Chen, B. Wu, "GPU accelerate parallel Odd-Even merge sort: An OpenCL method", *Computer Supported Cooperative Work in Design (CSCWD), 2011 15th International Conference*, pp. 78-83, (2011).

[18] C. Cerín, J.L. Gaudiot., "Parallel Sorting Algorithms with Sampling Techniques on cluster with Processors Runing at Different Speeds", *7th International Conference Bangalore, India*. pp. 301-309. (2000)

[19] V. Singh, V. Kumar, G. Agha, C. Tomlinson. "Efficient Algorithms for Parallel Sorting on Mesh Multicomputers", *International Journal of Parallel Programming*, Vol. 20 (2), (1991).

[20] S.S. Tseng, R.C. T. Lee, "A new parallel sorting algorithm based upon min-mid-max operations", *BIT Numerical Mathematics.* Vol. 24 (2), pp. 187-195. (1984).