Journal of Computer Science 4 (6): 447-453, 2008
ISSN 1549-3636
© 2008 Science Publications

Deterministic Parallel Sorting Algorithm for 2-D Mesh of Connected Computers

[1]Hamed Al Rjoub, [2]Arwa Zabian and [3]Ahmad Odat
[1,3]Irbid National University, Irbid-Jordan
[2]Jadara University, Irbid-Jordan

Abstract: Sorting is one of the most important operations in database systems and its efficiency can influences drastically the overall system performance. To accelerate the performance of database systems, parallelism is applied to the execution of the data administration operations. We propose a new deterministic Parallel Sorting Algorithm (DPSA) that improves the performance of Quick sort in sorting an array of size n. where we use p Processor Elements (PE) that work in parallel to sort a matrix r*c where r is the number of rows r = 3 and c is the number of columns c = n/3. The simulation results show that the performance of the proposed algorithm DPSA out performs Quick sort when it works sequentially.

Key words: Parallel sorting, deterministic algorithms, Quicksort, Parallel Quicksort

INTRODUCTION

Sorting is one of the most common operations in parallel processing applications. For example, it is central to many parallel data base operations and important area such as image processing, statistical methodology, search engine etc…. It is well known that sorting can be done with O (n log n) comparisons for Quick sort. This order can be reduced if we use the parallelism, where we can sort n elements in one round using 2^n processors to make all the comparison at once. A number of different types of parallel sorting scheme have been developed. Parallel deterministic sorting algorithms are based on Compare-and- Exchange (CE) operations. The lower bound on the number of CE operations to sort n element is Ω (nlogn). Several optimal sequential algorithms are known, such as Quicksort[4] and Heapsort[4]. A vast number of parallel sorting algorithms have been described in the literatures, some of them are cost-optimal PRAM parallel sorting algorithms[16]. The most fundamental one is called Cole's Merge Sort[5,14], it sorts n numbers using n processors in time O (logn). Others are synthetically optimal mesh sorting algorithms, in which for $N = n^2$ numbers can be sorted on N-2D mesh (n, n) in time kn where k is a constant between 2 and 3, n is the dimension of the mesh. The number of sorting steps is bounded by the diameter of M (n,n), which is 2n-2. The most important sorting algorithms for sorting n numbers on N-nodes hypercube networks is Batcher's Merge Sort[6] which needs $O(\log^2 n)$ steps.

There exists deterministic parallel algorithms for sorting n numbers on n-node hypercube network in O (log n (log log n)) CE steps but the hidden constant is very large.

Parallel Bubblesort[7], is an oblivious sorting on-mesh, it takes precisely n steps to sort n numbers on 1-D mesh which is optimal due to the diameter of the mesh. The idea is obvious, alternate CE operations between odd-even and even-odd transposition sort. The cost of the algorithm is O (n^2).

Even-odd transposition sorting algorithms for n>p numbers on p processors, it performs P sequential operations with n/p numbers and each operation takes θ (n/p) CE operation and the total parallel time is θ (n/p log n/p) + θ (n/p).

The scalability of even-odd transposition sort is therefore very poor, since to keep a constant efficiency, input data size must grow exponentially with the number of processors. Shearsort[8,9], it is based on the even-odd-transposition, it works for any 2-D mesh. Shearsort on M (n,n) consists of 2logn+1 phases and $\lceil \log n \rceil$ column phase to sort nm numbers. Each row (phase) takes θ (m) times and each column takes θ (n) time.

The goal of our work is to demonstrate that using Quicksort in parallel in sorting an array of size n, can reduce the sorting time significantly respect to the case where Quicksort sorts the same array. For that, we propose a deterministic parallel algorithm that sort (3, n/3) mesh in n/3 deterministic steps with running time is equal to O (n/3log n/3).

Deterministic Parallel Sorting Algorithm (DPSA):
Algorithm description: Given an array A of size n, our algorithm must be able to sort A in a polynomial time O

Corresponding Author: Hamed Al Rjoub, Irbid National University, Irbid-Jordan

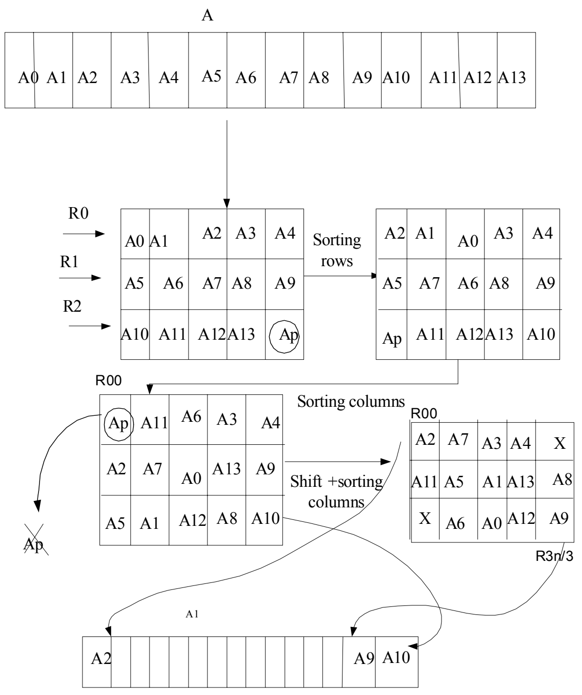

Fig. 1: The description of DPSA

(n/3logn/3).The array A id divided into a matrix R×C where R is the number of rows (R = 3) and C is the number of columns that must be an odd number calculated as follow C = n/3. If 1≤n mod 3≤3 then C = $\lfloor n/3 \rfloor$ +1 else C 0 = $\lfloor n/3 \rfloor$.

P_0, P_1, P_2 are three element processors will be assigned to R_0, R_1, R_2 (rows) respectively to sort them in parallel. Then P_0, P_1, P_2, P_c are element processors will be assigned to C columns to sort them in parallel. After each sorting operation to the columns, two sorted elements are generated and will be placed in parallel in the final array A^1 (Fig. 1).

The algorithm works in two phases:

First phase:

- To each element in A assign a Processor Elements (PE), have the role to perform the following operations: read, write (shift).
- Dividing the array A into three rows R_0, R_1, R_2 with fixed length C, C = n/3 here we have the following cases:
- If n mod 3 = 1 and C is an odd number, add two elements with highest digits to the last column (for example, if the biggest element of A is composed on three digits we add an element with 4 digits). These elements will be sorted and deleted initially.
- If n mod 3 = 2 and C is an odd number, add only one element with highest digit and will be treated as in the previous case.
- If C is not an odd number, add three elements (one column) with highest digit as in the first case (worst case).
- Use Quick sort to sort R_0, R_1 and R2 in parallel.

Second phase:

- Assign P_0, P_1, P_2 and P_c, to sort C columns in parallel
- When the sorting operation is finished, the elements in the first and last location in the matrix will be the maximum and minimum elements respectively in the array, these elements will be added to their location in the final array A^1 (that correspond the first and the last element in the array A^1).
- Shift the elements in the first row one location to the left and the elements in the last row on location to the right.

These two steps (2, 3) will be repeated until all the elements in the first and the last row (R_o, R_2) are sorted, that means the number of iterations will be n/3 iterations. As a natural result after we finished sorting and shifting operations for R_0, R_2 we obtained a sorted elements for the middle row R_1, which will be merged in parallel in one step to its correspondent location in A1.

Example: Figure 1, represent an array A of size n = 14, in the first phase (number1) A is divided to R_0, R_1, R_2 where n mod 3 = 2. That means an element A_p with the highest digit will be added to the final column of the array. C = 5 is an odd number. In the middle part of Fig. 1, it is clear that after the sorting rows operation and the sorting columns operation A_p is joined to the location R_{00} and will be sorted and deleted. In the second shifting and sorting operations (the last step in Fig. 1) it is clear that two sorted elements in the location R_{00}, $R_{3n/3}$ are added to the final array A^1 in parallel. Figure 2a, represents a numerical example with an array A of size n = 15. It is clear that after each sorting operation two elements are added to the final array A^1. Figure 2b shows that after the second shifting operation the middle element denoted by a circle will be stay in its location and never change it. Figure 2a and b, demonstrate that the sorting operations is equal to n/3 = 5 and is noted from Fig. 2b that after 5 iterations the middle row will be already sorted and will be added in parallel to its location in the final array $A^{1.}$

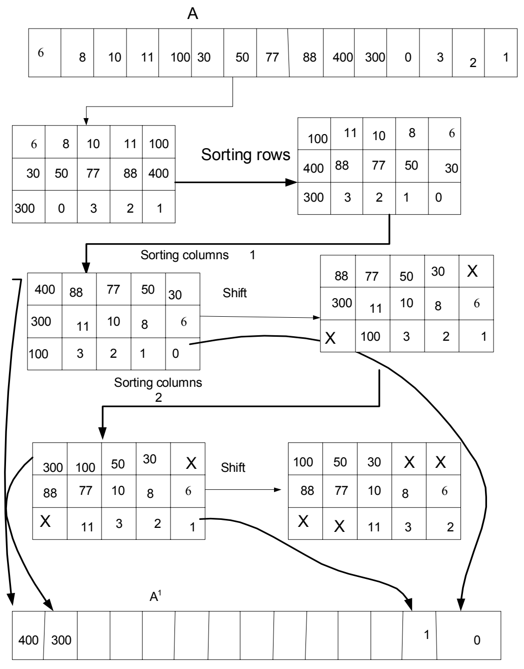
Fig. 2a: Numerical description of DPSA work: First phase

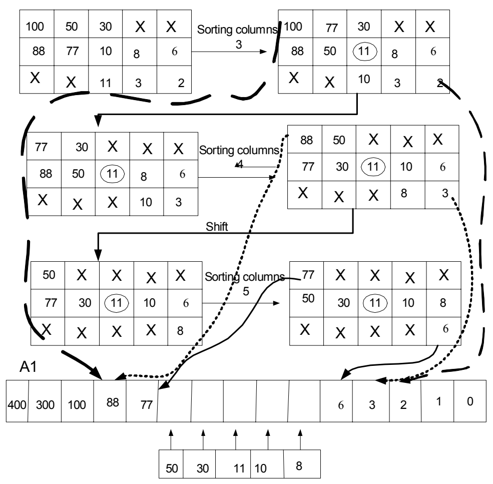
Fig. 2b: Numerical description of DPSA: Second phase

Complexity analysis: The first phase is done only one time its complexity is calculated as follow:

- Assigning n processor elements requires a time θ (1)
- Sorting rows has complexity n/3logn/3.
- So the complexity of the first phase is

$$T_1 = n/3 \log n/3 \quad (1)$$

The complexity of the second phase is calculated as follow:

- Sorting columns first time has complexity 3Clog3 where C is the number of columns
- Then the consecutive sorting operations has the following complexity (n-2j)log(n-2j) where j is the number of iterations is equal to n/3 and in each iteration the number of sorted elements will be reduced by 2 respect to the previous iteration. So, the time complexity of the second phase is:

$$T_2 = 3C\log 3+n/3\log n/3 \quad (2)$$

The complexity time is calculated as follow:

$$T = T_1+T_2 = n/3\log n/3+3C\log 3+ n/3\log n/3 \le O\ (n/3\log n/3) \quad (3)$$

That means, dividing the array to 3 rows has reduced the complexity of Quick sort in the average case from O(n log n) to O(n/3 log n/3), where the number of comparisons is reduced to n/3. This reduction is significant when the number of input becomes large.

Lemma: For a given array A of size n our algorithm sort A in a polynomial time and the number of steps needed is exactly n/3 steps. In which each element is sorted only once. And the resultant middle row with n/3 element is already sorted.

Proof: We will prove that the number of steps needed to sort n elements given our algorithm will be exactly n/3. For that we propose that the number of steps initially will be more than n/3. For example n/3+i, where i is an integer.

In our algorithm in each step after the sorting operation, two elements were sorted in parallel. That means, when the algorithm stop the number of sorted elements will be: 2(n/3+I) added to it the number of elements of the middle row that is already sorted n/3 that means the total number of element sorted will be:

$$2(n/3+ i) + n/3 = n +2i$$

However the number of element in the array is only n that is a contradictory. So, the number of steps cannot be n/3+i.

From our algorithm the sorted element must joins one of the following two locations: the first or the last element in the matrix (n/3, 3) and then will be inserted in its location in the final array A^1, that means each element cannot be sorted twice, consecutively if the array is n elements the number of sorted elements cannot be n+2i and the number of steps in our algorithm is exactly n/3.

RESULTS AND DISCUSSION

To evaluate the effectiveness of the parallelism in sorting an input data of size n, we have done a simulation using Microsoft Excel 2007, in which is calculated the running time of DPSA with the variation of input data size. Table 1 represents our results, where it is evident that the time needed for sorting is increased with the input size. However, the running time of DPSA is better than Quick sort for the same input size. Figure 1 shows that for an input size from 50-1000 items both DPSA and Quick sort have the same performance. However, when the input size grows, DPSA out performs the performance of Quick sort. That means, DPSA has reduced the running time of Quick sort about 2.06 times and that because we have reduced the number of comparisons to n/3.

The key issue in the parallel processing of a single application is the speedup achieved or the efficiency of parallel processing. That is defined as the factor by which the execution time for the application changes. That is:

Speedup (accelerator) = execution time for one processor / execution time for P processors.

Based on that, the efficiency is calculated as follow:

$$\text{Efficiency} = \frac{\text{Accelerator}}{\text{No. of processors}} \times 100$$

The ideal values for the efficiency must be 1/p that means dependent on the number of processors (p) in a complete parallel system (Table 2).

Table 2 shows that for small input size the efficiency of DPSA is far from the ideal however, when the input size increases the efficiency of DPSA is near ideal. In[10], is proposed a generic parallel sorting algorithm (PRMQ) that convert an array of size n to a matrix r*c where n = r*c in a manner that r is an odd number r≥ 3 and c≥3. Figure 3 shows that the performance of DPS is outperforming both Quick sort and PRMQ for the same input size and the same number of processors. However, a comparison between the efficiency of both PRMQ and DPSA with the ideal efficiency (1/p) in a complete parallel system is explained in Table 3. Figure 4 shows that DPSA has efficiency near ideal in comparison to PRMQ. While

Table. 1: The running time of Quick sort and DPSA given the input size

Size (n)	Quick sort	DPSA Sort	No. of col.
9	285.2932501	138.3721876	3
20	864.3856190	385.2939563	7
50	2821.9280950	1182.0562400	17
100	6643.8561900	2700.4458130	35
200	15287.7123800	6058.5582930	67
500	44828.9214200	17348.1092300	167
1000	99657.8428500	38026.5517800	333
1200	122745.8243000	46688.1997600	401
1400	146316.9556000	55504.8976900	467
1600	170301.6990000	64465.4663500	535
1800	194648.0614000	73546.5746500	603
2000	219315.6857000	82722.7702300	667

Table 2: The efficiency of DPSA

Size	P = n/3	Running time For one processor (ms)	Running time for DPSA (ms)	Efficiency	Ideal %
9	3	285.29	138.37	22.9%	33.0
20	7	864.38	385.29	11.2%	14.0
50	17	2821.92	1182.05	4.8%	5.0
100	34	6654.85	2700.44	2.5%	2.0
200	67	15287.71	6058.55	0.5%	1.4
2000	667	219315.60	82722.77	0.1%	0.1

Table 3: Comparison between the efficiency of DPSA and PRMQ given the ideal efficiency

Size	p	Efficiency PRMQ	Efficiency DPSA	Ideal 1/p
50	17	8.83%	4.8%	5%
100	34	4.67%	2.5%	2%
200	67	2.48%	0.5%	1.4%

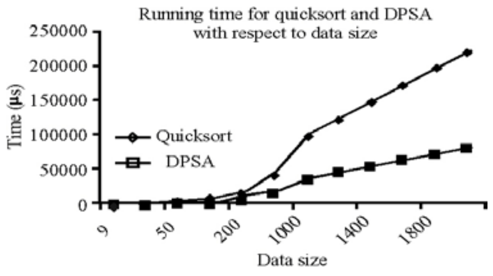

Fig. 3: A comparison between Quicksort and DPSA running time

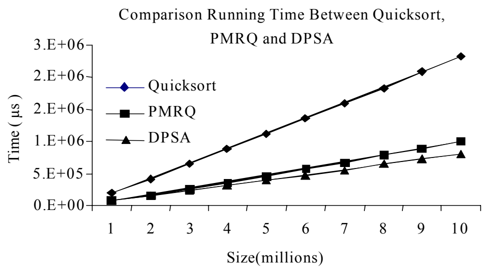

Fig. 4: Comparison between DPSA, PRMQ, Quicksort running time

Table. 4: The comparison between the running times of Quick sort, DPSA, and PRMQ algorithms for a large data size

Size(n)	Running Time Quick sort(μs)	Running Time DPSA(μs)	Running Time PMRQ(μs)
$1*10^6$	1.99E+05	7.12E+04	7.01E+04
$2*10^6$	4.19E+05	1.49E+05	1.58E+05
$5*10^6$	1.11E+06	3.95E+05	4.56E+05
$8*10^6$	1.83E+06	6.42E+05	7.82E+05
$9*10^6$	2.08E+06	7.36E+05	8.94E+05
$1*10^6$	2.33E+06	8.13E+05	1.01E+06

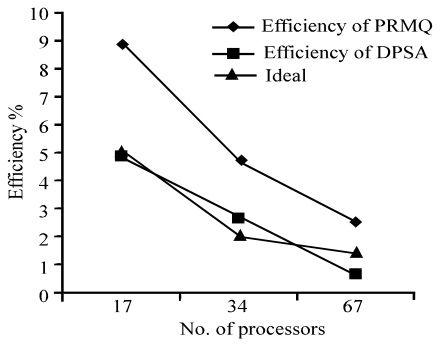

Fig. 5: Comparison between the efficiency of PRMQ and DPSA with the ideal values in completely parallel system

PRMQ is less efficiency than DPSA. Table 4, show that for a large size of input data DPSA running is time is better than both Quick sort and PRMQ.

CONCLUSION

In this study we have proposed a new parallel mechanism to sort an array of size n in a manner to reduce the number of comparisons. Using the parallelism has reduced the running time of Quick sort from O (nlogn) in the average case to O (n/3logn/3). In our work we have divided the array to a fixed number of rows. So, we have obtained a matrix (3, n/3) where the columns are sorted in parallel using P Processor elements. Our simulation results show that using Quick sort in parallel in the manner described in our algorithm (DPSA) has conducted to more efficient sorting in term of running time and number of comparisons.

Parallel sorting can form a basic building block to implement higher level combinatorial algorithms. In[3], is proposed a parallel sorting algorithm which moves a minimal amount of data over the network. Where it is used P processors to sort an array of size n. The algorithm works in four phases are: local sort, splitting, elements routing and merging phases. In the local sorting phase is applied any sorting algorithm for non parallel sorting to do the primary sorting in parallel in total cost $T_s = T_s(\lceil n/p \rceil)$ where T_s is the time needed by the algorithm used to sort n elements with O(n/p log n/p) comparisons.

The splitting phase is divided into three sub phases are: single selection, simultaneous selection and producing indices. In the second phase, the splitting phase is used the binary search rather than partition, where the elements are ranking using O (log n) rounds. In the single selection sub phase is selected only one element with global rank r. It is defined an active range with contiguous sequence elements have the rank r and each round the active range is divided in two and determined the target elements. In the simultaneous selection sub phase is selected multiple target with different global rank, in this sub phase the amount of data being sent is O(p^2 log n) over O(log n) round. In the third phase the elements routing phase, the elements are moved from the location they start out to where they belong. The best case if the elements are already sorted the worst case if the elements are reversed sorted order where it needs θ (n) steps. In the final phase, the merging phase is merged the p sorted sub vectors in a single sorted sequence, for this phase is used the binary tree where each elements moves at most $\lceil \log p \rceil$ times and the total time needed will be $\lceil n/p \rceil \lceil \log p \rceil$. The total computing time is $1/p\ T_s(n) + O(P2 \log n + p \log^2 n) + (\lceil n/p \rceil$ if p not power of 2). The simulation results show that the communication cost for sorting algorithm is near linear if p<<n. In[11], is studied the problem of parallel sorting on a two dimensional mesh multicomputer architecture, where it is used a new parameter to evaluate the performance of the algorithm that is the scalability. The scalability consists on comparing the performance of parallel algorithms when the data input size and the number of processor can vary. That means, is the ability of the algorithm to be efficient by using increasing number of processors. QSP_1[11]. Selects an element as the pivot, then the number of elements bigger and smaller than the pivot is counted. The smaller elements are moved to the processors that come first in the row-major ordering. And larger elements are moved to the processors that come later in the row-major ordering. This step is repeated recursively until a partition fits only within one processor. At that time, a sequential merge sort is used for sorting the local elements. The computation time is $\theta(2^{k(p)1/2 \log p}(p)^{1/.2} \log p) = \theta(n(p)^{1/2} \log p)$ where k is constant, p is the number of processors. QSP_2[11],is a variant of QSP_1 where the partitioning is done alternately in the vertical and horizontal dimensions in a manner that the maximum distance

within each partition is reduced by a factor of two after each set of one horizontal and one vertical partitioning. The overall complexity for all steps is θ (n (p) $^{1/2)}$ as apposed to θ (n (p) $^{1/2}$ log p) for QSP_1. The total number of comparisons in $QSP_{1,}$ QSP_2 no more than O (n log n).

Lang Sort[12], is a parallel sorting algorithm for SIMD Mesh parallel computers, where each processor has 1 element. It is based on exchange and compares exchange operation, where in the exchange operations two elements were exchanged between two processors. And in the compare-exchange operations two elements are exchanged if and only if the elements are not in correct sorted order. In[14], Batcher proposed Bitonic sorting networks algorithm that achieves non optimal parallel running time O ($\log^2$ n) for sorting a network with n nodes. A sequence is called Bitonic sequence if there is a value of j such that after rotation by j elements, the sequence consists of monotonic increasing part followed by a monotonic decreasing part. For j $\in$ {0, ……. n-1} the operation on the sequence { a_0,……..a_{n-1}} result on {a_j,…..a_{n-1},a_0 ,…………a_{j-1}} the rotation by j mod n.

Bitonic sorting algorithm transforms the bitonic sequence into its corresponding monotonic increasing (or monotonic decreasing) sequence. Then it merges the two sorted sequence. The merging operation (Bitonic merge) of the two sorted sequences can be performed as follows: if two sequences are sorted in opposite sorting directions the concatenation of two sequences yields a bitonic sequence. Thus the result of the transformation into a monotonic increasing sequence corresponds to the results of merging the two input sequence according to the respective sorting direction. If the two sequences are sorted in the same sorting direction one of them would have to be reversed. The transformation of the sequences or the bitonic merge is done recursively. In adaptive bitonic sorting algorithm[13] an index j* is defined as follow {-n/2 …n/2-1} where for a sequence a of n elements after rotation of a by j* elements, all elements of the first half (p) are not greater than any element of the second part (q). Adaptive bitonic merge algorithm is based on Min/Max determination algorithm that determines the minimum as well as the maximum components of the bitonic sequences p and q in O (log n) time. Adaptive bitonic sorting algorithm is based on Batcher's algorithm where the merge step is performed by reordering a bitonic sequence. It is an optimal parallel sorting algorithm, it runs in time O ($\log^2$ n) parallel time with O (n/log n) processors. GPU-ABISort basic algorithm proposed in[15], is based on adaptive bitonic sorting algorithm[13], where it is used the stream architecture instead using Min/Max determination algorithm in parallel. Where the input data is divided into streams then the Kernel performs computation on entire streams or on the substream to produce one or more streams as output. To apply the technique of adaptive bitonic sorting, the random access write have to be replaced by stream write without contiguous stream blocks as large as possible. GPU-ABISort has reduced the number of stream operations by a factor of O (log n) in corresponding to the adaptive technique implemented in adaptive bitonic sorting algorithm[13]. The sorting approach on stream processors it achieves the optimal complexity O ((nlogn)/p).

REFERENCES

1. Duseau, A.C., D.E. Culler, E.E Schauser and R.P. Martin, 1996. Fast parallel sorting under logp: experience with CM-5. IEEE Trans. Parallel Distributed Syst., 7: 791-805.
2. Culler, D.E. A. Dusseau, S.C. Goldestein, A. Krishnamurthy, S. Lunetta, T. Von Eicken and K. Yelick, 1993. Parallel programming in split-C. In: Proceedings of High Performance Networking and Computing. ACM/IEEE Conference on Supercomputing. Portland- Oregon United States. pp: 262-273. 15-19 Nov. 1993. ISBN: 0-8186-4340-4.DOI: 10.1109/SUPERC.1993.1263470
3. Cheng, D.R., V.B. Shah, J.R. Gibert and A. Edelman. 2007. A Novel Parallel Sorting algorithm for Contemporary Architectures. Int. J. Parallel Prog., PP:1-12.
4. Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein. 2002. Introduction to algorithms. Mc.Graw-Hill Higher Educatio. 2.Edition .
5. Cole, R., 1988. Parallel merge sort. SIAM J. Comput., 17: 770-788.
6. Morven, M., C. Meinel and D. Krob. 1998. STACS 98. Lecture Notes in computer Science 15th Annual Symposium on Theoretical Aspects of Computer science. Paris-France. February 25-27/1998. Springer. Springer; 1 edition (April 8, 1998). ISBN-10: 3540642307
7. Grama, A., A. Gupta, G. Karypis and V. Kumar, 2003. Introduction to Parallel Computing. Second Edition. January 26, 2003. Addison-Wesley ISBN-10: 0201648652
8. Knuth, D.E., 1973. The art of computer Programming. -Sorting and Searching. Addison-Wesley. 2nd edition (June 1973). ISBN-10: 020103803X

9. Scherson, I.D. and S. Sen, 1989. Parallel Sorting in Two dimensional VLSI Models of Computing. IEEE Trans. Comput., 38: 238-249.
10. Qawasmeh, S., A. Odat, H. and Al Rjoub, 2008. Parallel Matrix Representation- Quick Sort Algorithm (PRMQ). Accepted for Publication in The J. Comput. Sci.,
11. Singh, V. V. Kumar, G. Agha and C. Tomlinson, 1991. Scalability of parallel sorting on mesh multicomputer. In Proceedings of Parallel Processing Symposium. Anaheim, CA, USA. 30 April-2 May pp: 92-101. DOI: 10.1109/IPPS.1991.153762
12. Lang, H.W., M. Schimmler, H. Schmeck and H. Schroder, 1985. Systolic sorting on a mesh connected networks. IEEE Trans. Comput., 34: 652-658.
13. Bilardi, G. and A. Nicolau, 1989. Adaptive bitonic sorting: An optimal parallel algorithm for shared memory machines. SIAM J. Comput., 18: 216-228.
14. Batcher, K.E., 1968. Sorting networks and their applications. In: Proceedings of the 1968 spring Joint Computer conference (SJCC) 30 aprile-2May 1968. Atlantic City, NJ, USA, volume:32: PP:307-314.
15. Greb, A,and G. Zachmann, 2006. GPU-ABISort: Optimal parallel sorting on stream architectures. In: The 20th International Parallel and Distributed Processing Symposium IPDPS. 25-29 April 2006.PP:1-10. DOI: 10.1109/IPDPS.2006.1639284
16. Natvig, L., 1990. Logarithmic time cost optimal parallel sorting in not yet fast inpractice. In: Proceedings of Supercomputing'90. Nov. 12-16. New York, USA. pp: 486-494.