Communication-Efficient Bitonic Sort on a Distributed Memory Parallel Computer*

Yong Cheol Kim, Minsoo Jeon, Dongseung Kim
Dept. of Electrical Engineering
Korea University
Seoul, 136-701, Korea
dkim@classic.korea.ac.kr

Andrew Sohn
Dept. of Computer & Information Science
New Jersey Institute of Technology
Newark, NJ 07102-1982, USA
sohn@cis.njit.edu

Abstract

Sort can be speeded up on parallel computers by dividing and computing data individually in parallel. Bitonic sorting can be parallelized, however, a great portion of execution time is consumed due to $O(\log^2 P)$ *time of data exchange of N/P keys where P, N are the number of processors and keys, respectively.*

This paper presents an efficient way of data communication in bitonic sort to minimize the interprocessor communication and computation time. Before actual data movement, each pair processors exchange the minimum and maximum in its list of keys to determine what keys are to be sent to its partner. Very often no keys need to exchange, or only a fraction of them are exchanged. At least 20% or greater of execution time could be reduced on T3E computer in our experiments. We believe the scheme is a good way to shorten the communication time in similar applications.

1. Introduction

Sorting is one of the core computational algorithms used in many scientific and engineering applications. Many sequential sorts take $O(N\log N)$ time to sort N keys. Several parallel sorting algorithms such as bitonic sort[1], sample sort[6,7], column sort[5] and partitioned radix sort[10,14] have been devised to shorten the execution time. Parallel sorts usually need a fixed number of data exchange and merging operations. The computation time decreases as the number of processors grows. Since the time is dependent on the number of data each processor has, good load balancing is important. In addition, if interprocessor communication cost is not low such as in distributed memory computers, the amount of overall data to be exchanged gives a great impact on the total execution time.

* The work was partially supported by KRF grant no. KRF-99-041-E00287.

Parallel bitonic sort makes each processor maintain the same number of keys, thus, it balances its workload perfectly in each round. However, if it is implemented on a distributed memory parallel computer, a great portion of overall execution time should be paid for the interprocessor communication. It is because the algorithm demands to exchange all the keys each processor has throughout the *merge-and-split* steps. There have been some researches that reduce the amount of data exchange through remapping of keys in each processors by data layout[8], or comparing a parity defined by the number of "1" bits in its index[9]. But they can be applied only to shared-memory computers. We have enhanced the parallel bitonic sort on distributed memory computers by minimizing the number of keys to exchange, hence reduce the total sort time.

The paper is organized as follows. In section 2, we present the parallel bitonic sorting algorithm and the idea of efficient communication. Section 3 reports experimental results based on Cray T3E. The last section concludes this paper. Appendix is attached for performance evaluation.

2. Parallel Bitonic Sort

Generic bitonic sort

Bitonic sorting algorithm by Batcher [1] produces a sorted sequence after a few iterations of bitonic merge which converts two bitonic sequences of size m each to one monotonic sequence of size $2m$. A bitonic sequence is a sequence of numbers $a_0, \ldots, a_i, a_{i+1}, \ldots, a_{m-1}$ with the property that there exists an index i ($0 \le i \le m-1$), where a_0 through a_i is monotonically increasing and a_{i+1} through a_{m-1} is monotonically decreasing, or there exists a cyclic shift of indices so that the former condition is satisfied. Figure 1 shows a bitonic sorting network for 8 keys. Each vertical segment represents a 2-input 2-output comparator with its polarity which performs the *compare-and-exchange* operation. A comparator with downward (upward) arrow outputs the greater

0-7695-1153-8/01 $10.00 © 2001 IEEE

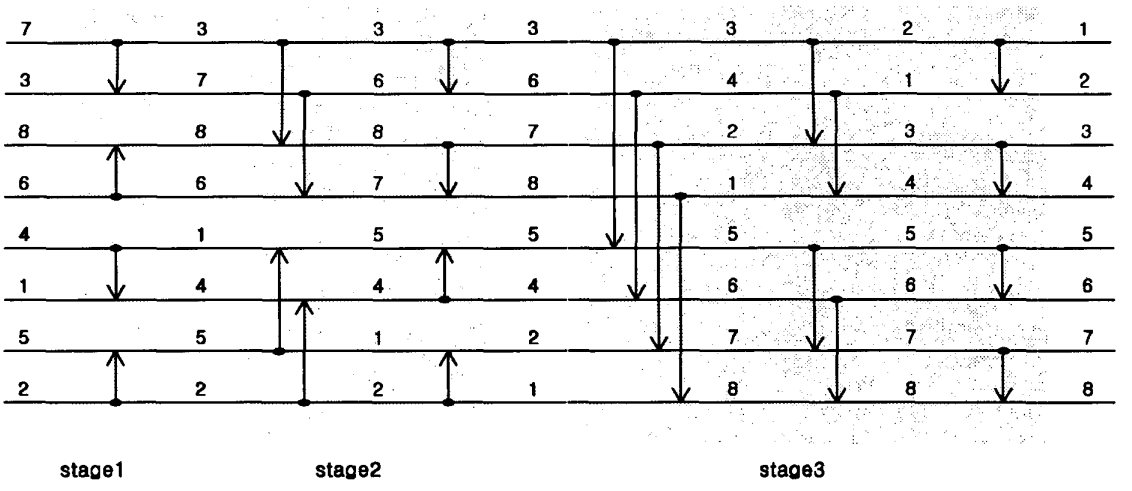

Figure 1. Bitonic sorting network for sorting 8 keys

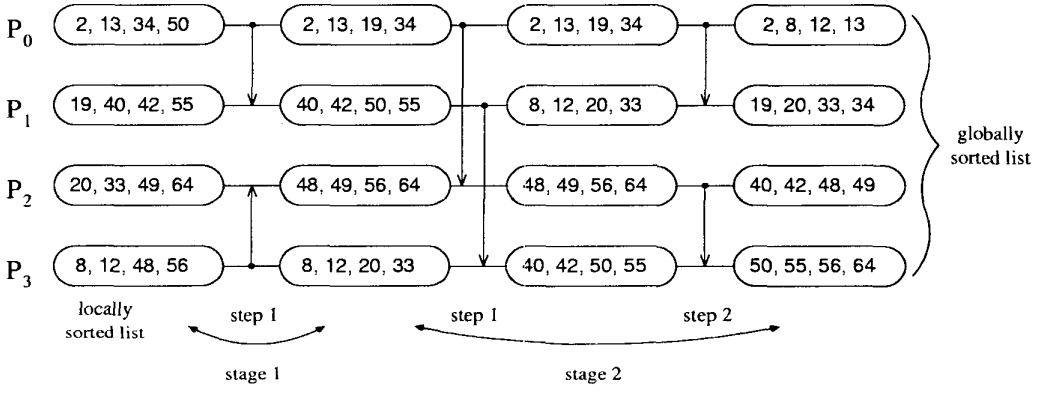

Figure 2. Parallel bitonic sort with 4 processors

value to its lower (upper) port, smaller to upper (lower). Bitonic sort with $m = 2^k$ input keys can sort them in time of $O(\log^2 m)$ by a network of $m(\log m)(\log m+1)/4$ comparators.

Mapping of bitonic sort algorithm to parallel computers can be done in a straightforward manner using the sorting network. Each comparator is replaced by a pair of processors with a merge-and-split operation, and data flow shown in the sorting network directs processors for their interaction in each step. Each processor is in charge of even number of $n = N/P$ keys, maintaining a sorted list of them throughout the sorting processes.

Each pair of processors performs *merge-and-split* operations in every step, in which two sorted sequences are admitted, merged into one monotonic sorted list, then, bisected into two (lower and higher) sequences. They correspond to the compare-and-exchange operations in the sorting network. After the merge-and-split, the two processors will keep the lower n and higher n keys respectively according to their polarity in the step.

Figure 2 illustrates the process of parallel bitonic sort of 16 keys with 4 processors. Initially each processor locally generates a sorted list of 4 keys. Then, parallel sort performs three steps of merge-and-split operations to complete. Each pair processors exchanges their lists, merges into one, then stores only a half to be used in the next step.

Execution time of the parallel sort can be analyzed below, where t_{local} is the time for the initial sort to make a sorted list,

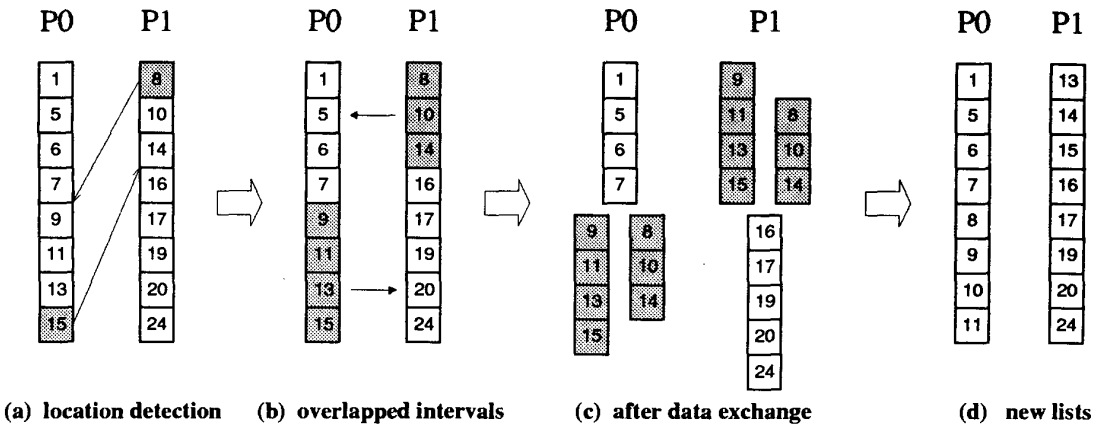

Figure 3. Merge-and-split for the partial exchange pattern

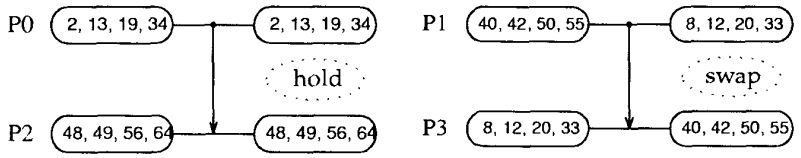

Figure 4. Hold and swap patterns

t_{comp} represents the time for merging two lists into a one, and t_{comm} is the interprocessor communication time to exchange their lists in a merge-and-split step.

$$t_{total} = t_{local} + \frac{(\log^2 P + \log P)}{2}(t_{comm} + t_{comp})$$

Reduction of communication in parallel bitonic sort

The point where we enhance the bitonic sort performance is the inefficient interprocessor communication in merge-and -split steps. At every merge-and-split step, a processor is supposed to receive a list of n keys from its partner processor to generate a newly sorted list of n keys. Instead of exchanging the whole n keys, our method reduces the number drastically by including only necessary keys. Detailed scheme is described below.

For a merge-and-split operation, a processor first accepts two sorted lists with n keys each from the paired processor, merges them, then bisects into two list, one in upper range and another lower. It keeps only one of them for next merge-and-split depending on its polarity, and discards the other. Note that each of the two lists has been already sorted before the exchange. A processor may not need the whole n keys from its partner since it eventually keeps the smallest (or the biggest) n keys among the merged $2n$ keys. It only needs a few keys sufficient to generate the n-key list. Refer to Figure 3 for an illustration(n=8). Here, P_0 and P_1 are assumed to be a pair of processors to interact for merge-and-split. The pair processors exchange their boundary (max, min) keys initially. The maximum key (15) of the list in P_0 lies between 8 and 24 of the list of P_1, and the minimum (8) of P_1 is between 1 and 15 of P_0. Then, their right positions in the lists are found by binary search (indicated in Figure 3a). Now, only the keys in the overlapped intervals (shaded area in Figure 3b) are transmitted to their partners. Merging is done in both processors to generate their combined lists. Now P_0 will select the smallest 8 keys whereas P_1 will keep the biggest 8 (Figure 3d). In this example, P_0 and P_1 send only 5(=3+2) keys to their respective partners,

instead of 16 (=8+8) for the case of generic bitonic sort. In summary, only the keys lying in the overlapped intervals are sent to their paired processors. This process is called *partial exchange*. The intervals are found by binary search using the boundary keys.

In the above process two exceptional cases occur that are dealt differently: 1) if there is no interval overlapped in the ranges of the two lists, the content of each list will not change. There may be no change of keys in both processors even after the merge-and-split, or 2) the two lists are completely switched to their partners. The former is the case of *hold*, where, once it is identified, no data are exchanged. In the latter case, rather than moving the whole lists, the owner's indexes (processor ids) are swapped, which is called *index swap*. The ids of swapped processors should be notified to other processors that will communicate with them in later steps/stages. Figure 4 shows an example of hold and index swap, redrawn from Figure 2 for convenience.

The enhancement of this sort can be achieved if the accumulated overhead does not exceed the gain due to the reduction in communication time. The scheme has to send boundary keys to each paired processor at each step, and a broadcast is needed for each occurrence of index swapping. If the size of the list is fine grained, the reduction of keys to be exchanged may not contribute to shorten the execution time. Thus, our scheme is effective when the number of keys is not too small to overcome the overhead.

3. Experimental Results

We have implemented the sorting algorithm in C language with MPI message passing library. The experiments have been performed on the distributed-memory CRAY T3E supercomputer that consists of 450 MHz Alpha 21164 processors and 3-D torus network. Keys are generated synthetically in each processor. The performance may differ depending on the distribution of keys. Three distribution functions are employed: *uniform, gauss, stagger* [14]. Multiple processors (P) up to 64 processors are used, and the size of input (N) ranges from 4K to 2M integers. Figures 5 and 6 show the improvement of the proposed parallel sort (*optimized*) over the generic bitonic sort (*generic*) with various key distributions. It is observed that the enhancement gets better as the number of processors grows, and is not very sensitive to key distribution. Sort of less than 8K keys results in no improvement on T3E due to too small size of input as mentioned before.

The ratios of the total merge-and-split time of the proposed sort to that of generic algorithm are given in Figure 7. The performance tends to increase as the number of processors as well as the key size per processor expands. The execution time is lowered by 3% (4 processors) to 43% (64 processors).

We have observed that the improvement gets better as the number of processor increases. The chance of having the three time-saving communication patterns is higher as the number of lists grows, for the range of key values are separated into many segments and more pairs of processors have their segments separated (with no overlap) in their ranges. Results shown in Figures 8 and 9 support the effect of increasing the processors in the sort. On T3E with 64 processors, up to 48% of execution-time reduction is achieved. A limited analysis for the performance is given in the appendix to estimate the maximum performance.

4. Conclusion

We have improved bitonic sort by optimizing the communication process of merge-and-split steps. Three communication patterns are identified to shorten the communication and the associated computation times. We have achieved a maximal reduction of 48% in total execution time when the input keys have uniform distribution, with 128M keys on 64-processor CRAY T3E. Also we have observed that the performance is not very sensitive to the key distribution. The improvement is also insensitive to the input size with the same number of processors, however, better improvement can be achieved as the number of processor is increased. One restriction is given on the number of key counts per processor, since the overhead in detecting overlapped intervals and broadcasting processor ids can not be compensated for sorting of too small number of keys. We expect the same idea can be applied to parallel implementation of similar merging algorithms.

Appendix

To estimate the performance of the new sorting method, merge-and-split steps are analyzed for the case of *uniform* key distribution. At each merge-and-split step, keys are exchanged, then the residing list and the received list of keys are merged, whose times are $t_{comm}(n)$ and $t_{comp}(n)$, respectively. Point-to-point message passing uses the wormhole communication, which is modeled as follows [16,17]:

$$t = t_s + \frac{m}{W}$$

where m is the size of a message, t_s is startup time, and W is bandwidth.

Merging of two lists of size n each takes a linear time in the worst case, which is written as follows:

$$t_{merge} = t_{comp}(2n) + t_{comm}(n)$$

where, $t_{comp}(n) = K_1 n$, $t_{comm}(n) = 2(t_s + \frac{n}{W}) = K_2 n + K_3$.

The parameters Ks are dependent on machine architecture.

The time spent in the whole merge-and-split steps of the *generic* bitonic sort is computed as follows:

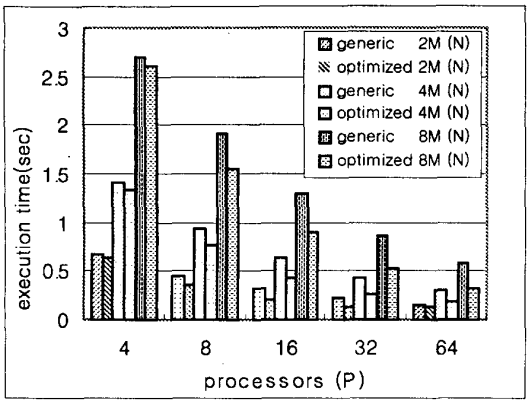

Figure 5. Execution times with *uniform* distribution

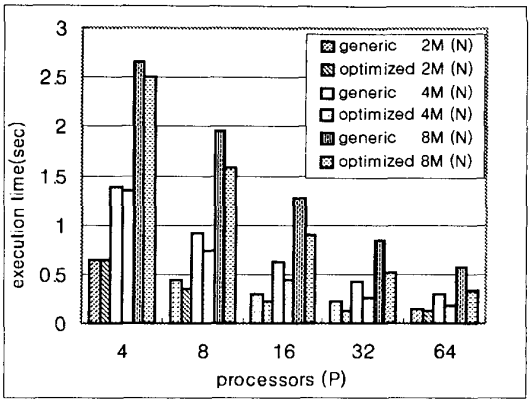

Figure 6. Execution times (*gauss*)

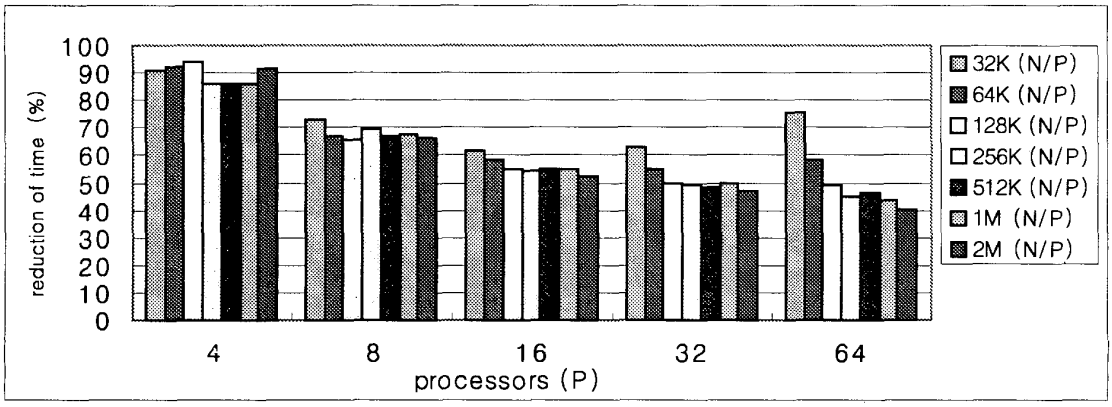

Figure 7. Comparison of execution times for split-and-merging stages

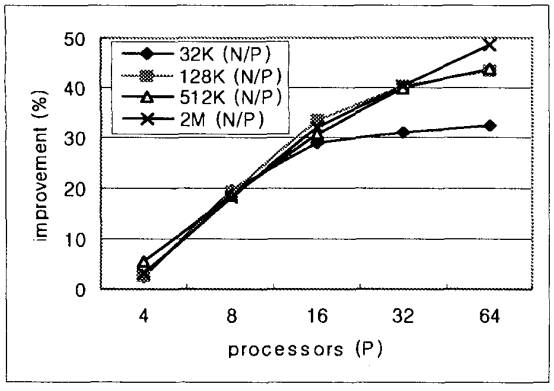

Figure 8. % improvement versus the number of processors

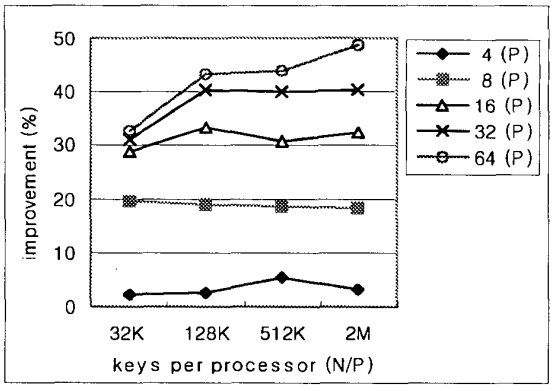

Figure 9. % improvement versus keys per processors

$$t_{generic.g.sort}(n,P) = \frac{1}{2}\log P(\log P+1)\{t_{comm}(n)+t_{comp}(2n)\}$$
$$= \frac{1}{2}\log P(\log P+1)(2K_1 n + K_2 n + K_3)$$

The time spent in the whole merge-and-split steps of the *optimized* sorting method is estimated as follows. In the bitonic merge of the sort, the size of sorted list grows 2, 4, 8, ..., 2^k at stages 1, 2, 3, ..., k, respectively. Merge stage p consists of p merge-and-split steps. With *uniform* distribution, only the last step in each stage tends to have severe data communication, and the rest need at most 10% of the overall keys. Thus, the execution time is estimated as

$$t_{enhanced.g.sort}(n,P) = \sum_{i=1}^{\log P}(i-1)\{t_{comm}(0.1n)+t_{comp}(1.1n)\}$$
$$+\log P\{t_{comm}(n)+t_{comp}(2n)\}$$

Enhancement of the new method over the generic one is obtained as

$$E_{g.sort} = \frac{t_{enhanced.g.sort}}{t_{generic.g.sort}}$$

For large n, K_3 can be ignored and the above equation can be simplified as follows:

$$E_{g.sort} = \frac{A\log P + B}{C(\log P+1)}$$

With the parameters of T3E all the constants in the equation can be computed as below:

$$t_s = 18\,\mu\text{sec},\quad W = 167\,\text{MB/sec}$$

$$K_1 = 1.98\times10^{-7},\quad K_2 = \frac{16}{W} = 9.6\times10^{-8},\quad K_3 = 2t_s = 3.6\times10^{-5}$$

$$A = 0.55K_1 + 0.05K_2 = 1.14\times10^{-7}$$
$$B = 1.45K_1 + 0.95K_2 = 3.78\times10^{-7}$$
$$C = K_1 + 0.5K_2 = 2.46\times10^{-7}$$

The equation says that if a number of processors are used in our parallel sort, the worst-case improvement reaches at the constant (A /C = 0.46) determined by the communication and computation parameters of the computer.

References

[1] K. E. Batcher, "Sorting networks and their applications", *Proc. AFIPS Conf.*, 1968, pp. 307-314.

[2] G. Baudet, and D, stevenson, "Optimal sorting algorithms for parallel computers", *IEEE Trans. Computers*, vol. C-27, 1978, pp. 84-87.

[3] G. E Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. zagha. "A comparison of sorting algorithm for the connection machine CM-2", *Proc. ACM Symposium on parallel Algorithm and Architecture*, July 1991, pp. 3-16.

[4] R. Diekmann, J. Gehring, R. L ling, B. Monien, M. N bel, and R. Wanka, "Sorting large data sets on a massively parallel system", *Proc. 6th IEEE Symposium on Parallel and Distributed Processing (SPDP) Conf.*, 1994, pp. 2-9.

[5] A. C. Dusseau, D. E. Culler, K. E. Schauser, and R. P. Martin, "Fast parallel sorting under LogP: experience with the CM-5", *IEEE Trans. Computers*, Vol. 7, Aug. 1996.

[6] D. R. Helman, D. A. Bader, and J. JaJa, "Parallel algorithm for personalized communication and sorting with an experimental study", *Proc. ACM Symposium on Parallel Algorithms and Architecture*, June 1996, pp. 211-220.

[7] J. S. Huang ad Y. C. Chow, "Parallel sorting and data partitioning by sampling", *Proc. 7th Computer Software and Applications Conf.*, Nov. 1983, pp. 627-631.

[8] M. F. Ionescu and K. E. Schawer, "Optimizing parallel bitonic sort", *Proc. 11th Int'l Parallel Processing Symposium*, 1997.

[9] J. D. Lee and K. E. Batcher, "Minimizing communication in the bitonic sort", *IEEE Trans. Parallel and Distributed Systems*, Vol. 11, No. 5, May 2000.

[10] S. J. Lee, M. S. Jeon, D. S. Kim, "Partitioned parallel radix sort", *Proc. Third International Symposium (ISHPC) 2000*, Tokyo, Japan, Oct. 16-18, 2000.

[11] F. T. Leighton, "Tight Bounds on the Complexity of parallel sorting", *IEEE Trans. Computers*, 1985, C-34: pp. 344-354.

[12] D. E. Muller and F. P. Preparata, "Bounds and complexities of networks for sorting and switching", *J. ACM*, 1975, vol.22(2), pp. 195-201.

[13] A. Sohn, Y. Kodama, M. Sato, H. Sakane, H. Yamada, S. Sakai, Y, Yamaguchi, "Identifying the capability of overlapping computing with communication", *Procs. ACM/IEEE Parallel Architecture and Compilation Technique*, Oct. 1996.

[14] A. Sohn, Y. Kodama, "Load balanced parallel radix sort", *Proc. 12th ACM Int'l Conf. Super Computing*, July 14-17, 1998.

[15] B. Wang, G. Chen, and C, Hsu, "Bitonic Sort with an Arbitrary Numbers of Keys", *Proc. 1991 Int'l Conf. Parallel Processing*, 1991, Vol. 3, pp. 58-61.

[16] L. M. Ni and P. K. McKinley, "A survey of wormhole routing techniques in direct networks", *IEEE Computer*, Feb. 1993, Vol. 26, pp. 62-76.

[17] R. Hockney, "Performance parameters and benchmarking of supercomputers", *Parallel Computing*, Dec. 1991, Vol. 17, No. 10 & 11, pp. 1111-1130.