Load-Balanced Parallel Merge Sort on Distributed Memory Parallel Computers *

Minsoo Jeon and Dongseung Kim
Dept. of Electrical Engineering, Korea University
Seoul, 136-701, Korea
{ msjeon, dkim}@classic.korea.ac.kr

Abstract

Sort can be speeded up on parallel computers by dividing and computing data individually in parallel. Merge sort can be parallelized, however, the conventional algorithm implemented on distributed memory computers has poor performance due to the successive reduction of the number of active (non-idling) processors by a half, up to one in the last merging stage. This paper presents load-balanced parallel merge sort algorithm where all processors participate in merging throughout the computation. Data are evenly distributed to all processors, and every processor is forced to work in merging phase. Significant enhancement of the performance has been achieved. Our analysis shows the upper bound of the speedup of the merge time as $(P-1)/\log P$*. We have had a speedup of 9.6 (upper bound is 10.5) on 32-processor Cray T3E in sorting of 4M 32-bit integers. The same idea can be applied to parallellize other sorting algorithms.*

1. Introduction

Sorting is one of the core computational algorithms used in many scientific and engineering applications. Many sequential sorting algorithms consume $O(N \log N)$ time to sort N keys. Several parallel sorting algorithms such as bitonic sort[1, 7], sample sort[10, 6], column sort[3] and partitioned radix sort[8, 11] 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 cheap such as in distributed memory computers, the amount of overall data to be exchanged and the frequency of communication give a great impact on the total execution time.

* This research was supported by KRF grant (no. KRF-99-041-E00287) and in part by KOSEF grant (no. R01-2001-00341).

Merge sort is frequently employed in many applications. Parallel merge sort on PRAM model was reported to have fast execution time of $O(\log N)$ for N input keys using N processors[2]. However, distributed-memory based parallel merge sort is slow because it needs local sort followed by a fixed number of iterations of merge that includes lengthy communication. The major drawback of the conventional parallel merge sort is that load balancing and processor utilization become poorer as it iterates: in the beginning every processor participates in merging its list of N/P keys with its partner's, producing a sorted list of $2N/P$ keys, where N and P are the number of keys and processors, respectively. In the next step and on, only a half of the processors used in the previous stage participate in merging process. It results in low utilization of resource (processor). Consequently, it lengthens the computing time.

This paper introduces a new parallel merge sort scheme, called *load-balanced parallel merge sort*, that forces every processor to participate in merging throughout the iterations. Each processor deals with a list of size about N/P at every iteration, thus the load of processors is well balanced which gives performance improvement.

The paper is organized as follows. In sections 2 and 3, we present the conventional and improved parallel merge sort algorithms together with the idea how more parallelism is obtained. Section 4 includes a performance analysis and section 5 reports experimental results performed on Cray T3E, Alpha11, and PC cluster. Conclusion is given in the last section.

2 Parallel Merge Sort: Conventional

Parallel merge sort consists of two phases: local sort and merge. Once keys in each processor are sorted locally, processors merge them in $\log P$ steps as explained below if P processors are used. In the first step, processors are paired as (sender, receiver). Each sender sends its list of N/P keys to its partner (receiver), then the two lists are merged by each receiver to form a sorted list of $2^1 N/P$ keys. A half of the processors work during the merge, and the other half

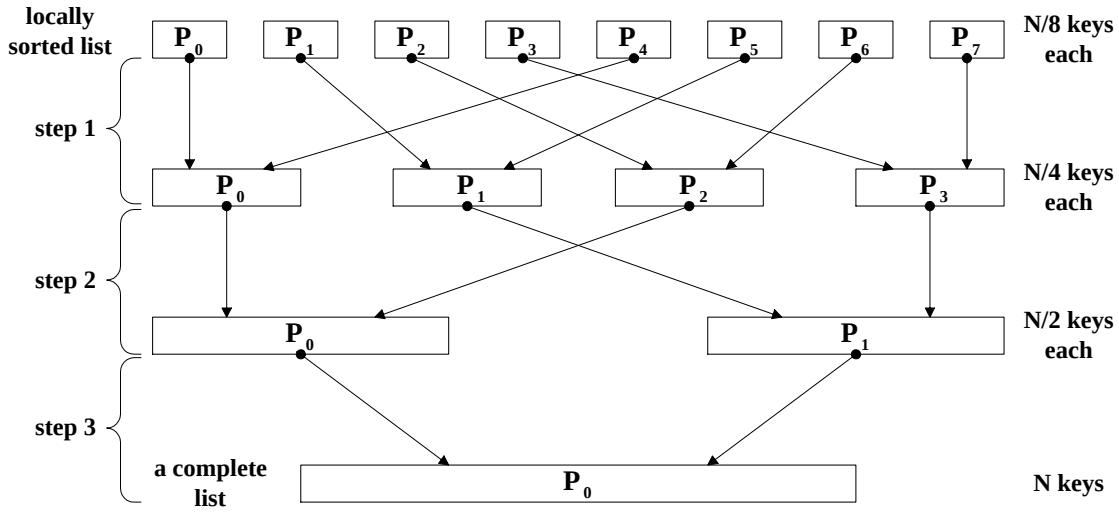


Figure 1. Conventional parallel merge sort with 8 processors

sit idling. In the next step only the senders in the previous step are paired as (sender, receiver), and the same communication and merge operations are performed by each pair to form a list of $2^2N/P$ keys. The process continues until a complete sort list of N keys is obtained. Figure 1 illustrates the conventional parallel merge sort of N keys with 8 processors. The algorithm is given as follows:

Algorithm 1: Conventional parallel merge sort
P: the total number of processors
(assume $P = 2^k$ for simplicity.)
P_i: a processor with index i
h: the number of active processors
begin
$h := P$
1. **forall** $0 \leq i \leq P - 1$
P_i sorts a list of N/P keys locally.
2. **for** j =0 **to** (log P)-1 **do**
forall $0 \leq i \leq h - 1$
if ($i < h/2$) **then**
2.1. P_i receives N/h keys from $P_{i+h/2}$
2.2. P_i merges two lists of N/h keys into a sorted list of $2N/h$
else
2.3. P_i sends its list to $P_{i-h/2}$
$h := h/2$
end

As mentioned earlier, the algorithm does not fully utilize the power of all processors. Simple calculation reveals that only P/log $P(= \{(P/2 + P/4 + P/8 + \cdots + 1)$/(log P steps)$\}$) processors are used in average per merging step. It must have inferior performance to an algorithm that makes a full use of them, if any.

3 Load-Balanced Parallel Merge Sort

To keep each merged list in one processor is simple and easy to handle as long as the algorithm is concerned. However, as the size of the lists grows, sending them to other processors for merge is time consuming, and processors that do not have lists after transmission sit idling until the end of the sort. The main idea in our parallel implementation is to distribute each (partially) sorted list generated after merge onto multiple processors such that each processor stores an approximately equal number of keys, and all of them take part in merging throughout the execution (see Figure 2). It would invoke more parallelism, thus shorten the sort time. One difficulty in this method is to find how to merge two lists each of which is stored not in a single processor, but distributed in multiple processors. We devise a way which minimizes the key movement as described below.

A *group* is a set of processors that are in charge of one sorted list. Each group stores a sorted list of keys by distributing them evenly to all processors and they keep non-decreasing (or nonincreasing) order according to the keys they store. In the first merging step, all groups have a size of one processor, and each group is paired with another group (called the *partner group*). In this step, there is only one communication partner per processor. Each pair exchanges their boundary keys (a minimum and a maximum keys) and determines a new order of the two processors such that one will keep the lower half of the merged list, the other the upper. Now each paired processor checks if there is any overlapped interval in their range of key values. (For ex-

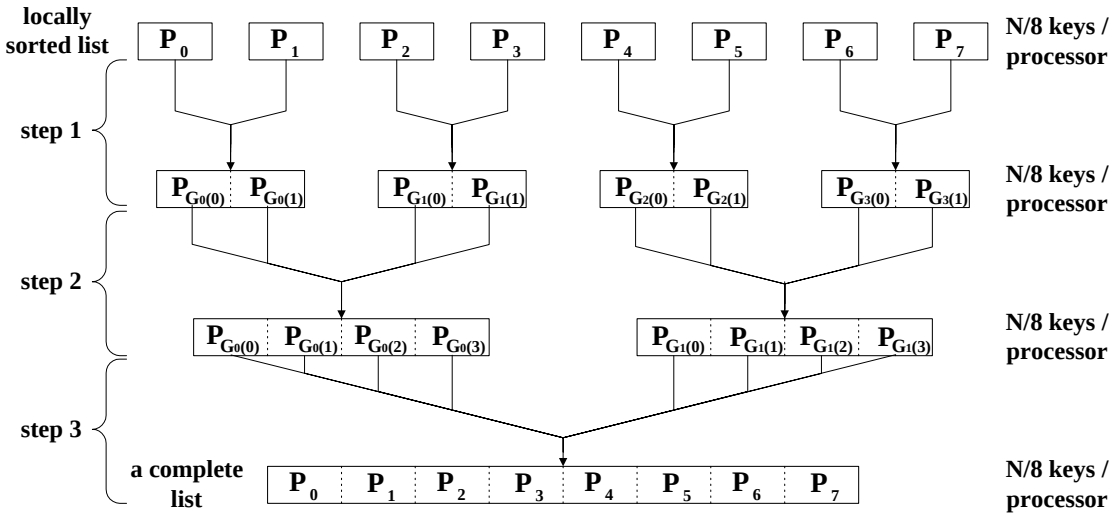


Figure 2. Load-balanced parallel merge sort

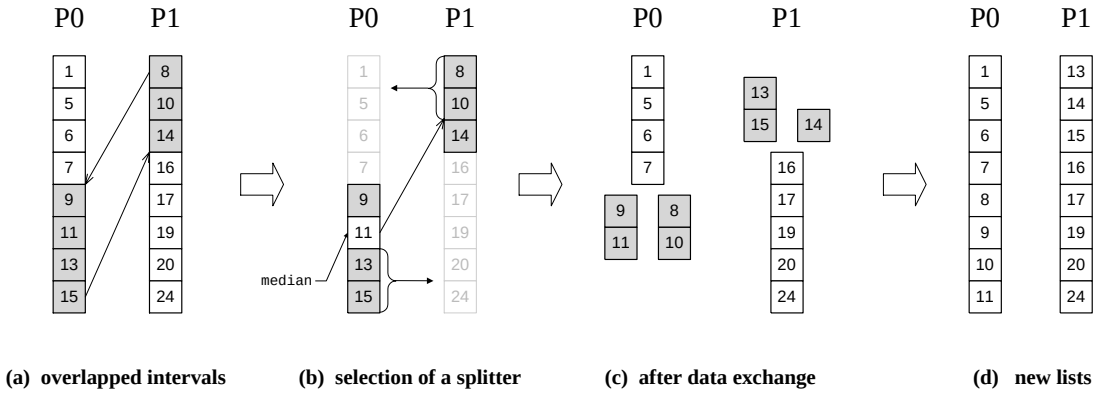


Figure 3. Process of detecting overlapped interval and merging that minimize data communication

ample, the two processors find those keys shaded in Figure 3a.) If yes, sequential merge is needed for keys lying within the interval. If we can find a *median* of the resultant list, only keys less than or equal to the median are needed in the lower half of the list, and those greater than the median in the upper half. So each pair of processors finds a *splitter*, which is an approximate median of all keys of the list. (For example, the key 11 in Figure 3b is the splitter.) Then, each pair exchanges keys in the overlapped interval so that one will keep all keys less than the splitter, the other greater. In this process at most $N/(2P)$ keys are exchanged. Now each processor holds $N/P \pm \Delta$ keys because the splitter may not be an exact median (we hope Δ is relatively small compared to N/P). Only the keys in the overlapped interval and those received from the sender need to merge (see Figure 3c,d). However, keys in the non-overlapped interval(s) do not interleave with keys of the partner processor's for merge. They are simply placed in a proper position in the final list. Often there maybe no overlapped interval at all, which is the case that no key exchange is needed.

From the second step and on, the group size grows twice the previous one. Each sorted list is divided and stored in a group of processors. Merging process is the same as above except that each processor may have multiple communication partners, up to the group size in the worst case. Now boundary values are again exchanged between paired groups and each processor merges keys of the overlapped intervals. The order of processors of the merged list is determined according to the order of the boundary values of processors (in each pair groups). Merging two groups into one may requires many processors to move keys. In order not to sequentially propagate keys of processors to multiple processors, only ids of the corresponding processors are swapped to have a correct sort sequence if needed. This minimizes the amount of keys exchanged among processors. If key values are random (i.e. uniformly distributed),

boundary values are good enough to determine the partners and the associated splitters. If not, global histogram of all keys aids the decision on which are communication partners and their splitters. The procedure of the merging phase is summarized as follows:

Algorithm 2: Load-balanced parallel merge sort
1. Each processor sorts a list of N/P keys locally.
2. Iterate log P times the following computation:
   /*At ith iteration, there are $P/2^{i-1}$ groups, each of which includes 2^{i-1} processors. */
   2.1. Each group of processors finds its pair (partner group).
   2.2. Exchange boundary values between paired groups.
   2.3. Determine overlapped interval(s) and communication partner(s).
   2.4. Determine the logical ids of processors that give a correct sequence of keys in the merged list.
   2.5. Perform the following with each communication partners.
      2.5.1. Find a splitter in each overlapped interval by binary search.
      2.5.2. Exchange keys lying in overlapped intervals.
      2.5.3. Merge keys, and make a sorted list which has about N/P keys per processor.
   2.6. Broadcast logical ids of processors.

Rather involved operations are added in the algorithm in order to minimize the key movement since the communication in distributed memory computers is costly. The scheme has to send boundary keys to each paired processor at each step, and a broadcast for the logical processor ids is needed before a new merging iteration. If the size of the list is fine grained, the increased parallelism may not contribute to shortening the execution time. Thus, our scheme is effective when the number of keys is not too small to overcome the overhead.

4 Performance Analysis

Let $T_{seq}(N/P)$ be the time for the initial local sort to make a sorted list. $T_{comp}(N)$ represents the time for merging two lists, each with $N/2$ keys, and $T_{comm}(M)$ is the interprocessor communication time to transmit M keys. For the input of N keys, $T_{comm}(N)$ and $T_{comp}(N)$ are estimated as follows[5]:

$$T_{comm}(N) = S + K_1 \cdot N \tag{1}$$

$$T_{comp}(N) = K_2 \cdot N \tag{2}$$

where K_1 and K_2 are the average time to transmit one key and the average time per key to merge N keys, respectively, and S is the startup time. The parameters Ks and S are dependent on machine architecture. For Algorithm1, step1 requires $T_{seq}(N/P)$. Step 2 repeats $\log P$ times, so execution time of the conventional parallel merge sort (CM) is estimated as below:

$$\begin{aligned} T_{CM}(N,P) &= T_{seq}(\frac{N}{P}) \\ &\quad + \sum_{i=1}^{\log N} \{T_{comm}(\frac{2^{i-1}N}{P}) + T_{comp}(\frac{2^i N}{P})\} \\ &\approx T_{seq}(\frac{N}{P}) + \{T_{comm}(\frac{N}{P} + \frac{2N}{P} + \cdots + \frac{\frac{P}{2}N}{P}) \\ &\quad + T_{comp}(\frac{2N}{P} + \frac{4N}{P} + \cdots + \frac{PN}{P})\} \\ &= T_{seq}(\frac{N}{P}) + \{T_{comm}(\frac{N}{P}(P-1)) \\ &\quad + T_{comp}(\frac{2N}{P}(P-1))\} \end{aligned} \tag{3}$$

In Eq.(3) the communication time was assumed proportional to the size of data by ignoring the startup time (Coarse-grained communication in most interprocessor communication networks reveal such characteristics). For Algorithm 2, step 1 requires $T_{seq}(N/P)$. The time required in steps 2.1 through 2.4 is ignorable if the count of keys per processor is large enough. Since the maximum number of keys in the overlapped interval in each processor is N/P, so at most N/P keys are exchanged among paired processors in step 2.5. Each processor merges $N/P + \Delta$ keys. Step 2.6 requires $O(\log P)$ time. Since step 2 is repeated $\log P$ times. The communication of steps 2.1 through 2.4 and 2.6 can be ignored since the time is relatively small compared to the communication time in step 2.5 if N/P is large (coarse grained). The execution time of the load-balanced parallel merge sort (LBM) can be estimated as below:

$$\begin{aligned} T_{LBM}(N,P) &= T_{seq}(\frac{N}{P}) + \log P \cdot \{T_{comm}(\frac{N}{P}) \\ &\quad + T_{comp}(\frac{N}{P} + \Delta)\} \end{aligned} \tag{4}$$

To observe the enhancement in em merging phase only, the first terms in Eqs. (3) and (4) will be removed. Using the relationship in Eqs. (1) and (2), merging times are rewritten as follows:

$$T_{CM}(N,P) = K_1 \cdot \frac{N}{P}(P-1) + K_2 \cdot \frac{2N}{P}(P-1) \tag{5}$$

$$T_{LBM}(N,P) = K_1 \cdot \frac{N}{P}\log P + K_2 \cdot (\frac{N}{P} + \Delta)\log P \tag{6}$$

A speedup of the load-balanced merge sort over the conventional merge sort, denoted as η, is defined as the ration of T_{CM} to T_{LBM}:

Table 1. Machine parameters

	K_1 [msec/key]	K_2 [msec/key]	C
T3E	0.048	0.125	1.732
Alpha11	0.029	0.064	1.691
PC cluster	0.386	0.083	1.184

$$\begin{aligned} \eta &= \frac{T_{CM}(N,P)}{T_{LBM}(N,P)} \\ &= \frac{K_1 \cdot \frac{N}{P}(P-1) + K_2 \cdot \frac{2N}{P}(P-1)}{K_1 \cdot \frac{N}{P}\log P + K_2 \cdot (\frac{N}{P} + \Delta)\log P} \end{aligned} \quad (7)$$

If the load-balanced merge sort keeps load imbalance small enough to ignore Δ, and N/P is large, Eq. (7) can be simplified as follows:

$$\begin{aligned} \eta &= \frac{K_1 \cdot \frac{N}{P}(P-1) + K_2 \cdot \frac{2N}{P}(P-1)}{K_1 \cdot \frac{N}{P}\log P + K_2 \cdot \frac{N}{P}\log P} \\ &= \frac{K_1 + 2K_2}{K_1 + K_2} \cdot \frac{P-1}{\log P} \\ &= C \cdot \frac{P-1}{\log P} \end{aligned} \quad (8)$$

where C is a value determined by the ratio of the interprocessor communication speed to computation speed of the machine as defined below

$$C = \frac{K_1 + 2K_2}{K_1 + K_2} = 1 + \frac{K_2}{K_1 + K_2} = 1 + \frac{1}{\frac{K_1}{K_2} + 1} \quad (9)$$

5 Experimental Results

The load-balanced merge sort has been implemented on three different parallel machines: Cray T3E, Alpha11, and PC cluster. T3E consists of 450 MHz Alpha 21164 processors and 3-D torus network. Alpha11 is a Linux cluster of 667MHz Alpha 21264 processors interconnected by Myrinet. PC cluster is a set of 8 PCs with 1GHz Athlon CPUs interconnected by a 100Mbps Fast Ethernet switch. We run the sorting programs onto up to 32 processors. Maximum number of keys is limited by the capacity of the main memory of each machine. Keys are synthetically generated with $uniform$ distribution [Sohn98] with 32-bit integers for Alpha11 and PC cluster, and 64-bit integers for T3E. Code is written in C language with MPI communication library [MPI].

Parameters of the computation and communication performance of individual systems are measured as given Table 1. The predicted and measured speedups of T3E, Alpha11, and PC cluster are recorded in Table 2. Note that T3E is expected to achieve the highest performance enhancement because of having the biggest C. Most of the results are close to the predicted ones except Alpha 11, for which the overhead (steps 2.1 to 2.4 in Algorithm 2) is not negligible due to the relatively slow network.

The speedups in merge time of the load-balanced merge sort over the conventional merge sort are shown in Figure 4. The improvement gets greater as the number of processors increases. The measured speedups are close to the predicted ones when N/P is large. When N/P is small, the performance suffers due to the overhead such as exchanging boundary values and processor id broadcasting. Experimental results support the fact given in Eq. (8) that among the three machines T3E has the highest speedup, Alpha next, and PC cluster the lowest.

The comparisons of the total sorting time of the load balanced merge sort with the conventional algorithm are shown in Figure 5. Local sorting times of both methods remain same in one machine. Figure 6 shows the speedup of the total sort time of the load-balanced merge sort over that of the conventional merge sort on three machines. The best speedups of 9.6 and 5.0 in merging phase and in total sort time respectively are achieved on T3E.

6 Conclusion

We enhanced the performance of parallel merge sort by distributing lists of keys to all processors, then merging by all processors throughout the computation. Logical ids of processors are modified and broadcasted in each iteration, and only keys in the overlapped interval are swapped to minimize the data movement. We have achieved a maximal speedup of 9.6 in merging time for 4M keys on 32-processor Cray T3E. The expected improvement of $(P-1)/\log P$ was observed in the experiments. Our algorithm is good if the distribution of keys is random/uniform and the number of keys per processor is not too small. This scheme can be applied to parallel implementation of similar merging algorithms such as quicksort.

References

[1] K. Batcher, "Sorting networks and their applications," *Proceedings of the AFIPS Spring Joint Computer Conference 32*, Reston, VA, 1968, pp.307-314.

[2] R. Cole, "Parallel merge sort," *SIAM Journal of Computing*, vol. 17, no. 4, 1998, pp.770-785.

[3] 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.

[4] W. D. Frazer and A. C. McKellar, "Samplesort: A sampling approach to minimal storage tree sorting," *Journal of the ACM 17*, 1970, pp.496-507.

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

Table 2. Comparison of predicted and measured speedups on T3E, Alpha11, and PC cluster with 4M keys

	P	2	4	8	16	32
T3E	$\eta_{predicted}$	1.732	2.585	4.02	6.461	10.683
	$\eta_{measured}$	1.691	2.660	3.661	6.379	9.595
Alpha11	$\eta_{predicted}$	1.691	2.537	3.946	6.341	-
	$\eta_{measured}$	1.634	1.960	2.667	3.497	-
PC cluster	$\eta_{predicted}$	1.184	1.776	2.763	-	-
	$\eta_{measured}$	1.142	1.601	2.301	-	-

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

[7] Y. Kim, M. Jeon, D. Kim, and A. Sohn, "Communication-Efficient Bitonic Sort on a Distributed Memory Parallel Computer", *Int'l Conf. Parallel and Distributed Systems (ICPADS'2001)*, June 26-29, 2001.

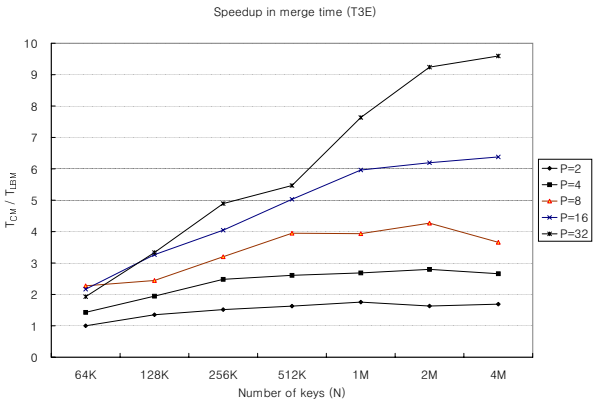
[8] S. J. Lee, M. Jeon, D. Kim, and A. Sohn, "Partitioned Parallel Radix Sort," *J. of Parallel and Distributed Computing*, Academic Press, (to appear) 2002.

[9] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," *Technical Report*, University of Tenessee, Knoxville, TN, June 1995.

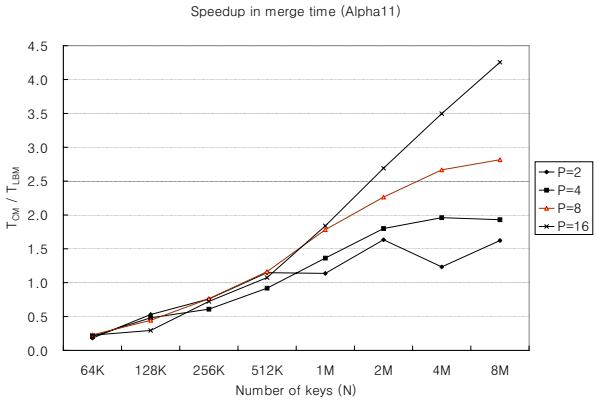
[10] H. Shi and J. Schaeffer, "Parallel sorting by regular sampling," *Journal of Parallel and Distributed Computing 14*, 1992, pp.361-372.

[11] A. Sohn and Yuetsu Kodama, "Load Balanced Parallel Radix Sort," *Proceedings of the 12th ACM International Conference on Supercomputing*, July 1998.

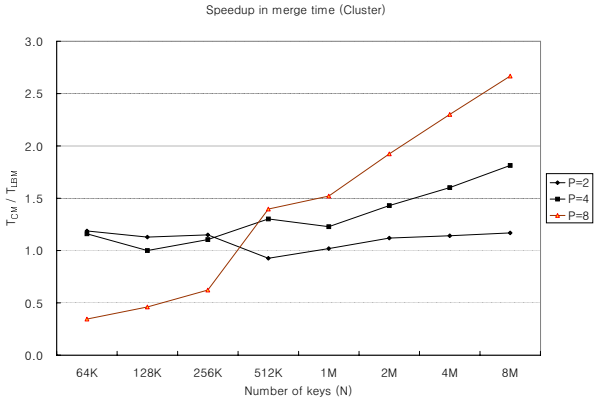
[12] R. Xiong and T. Brown, "Parallel Median Splitting and k-Splitting with Application to Merging and Sorting," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 5, May 1993, pp.559-565.



(a) T3E

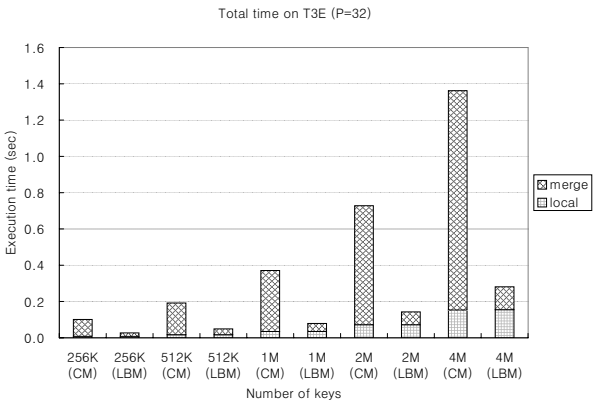


(b) Alpha11

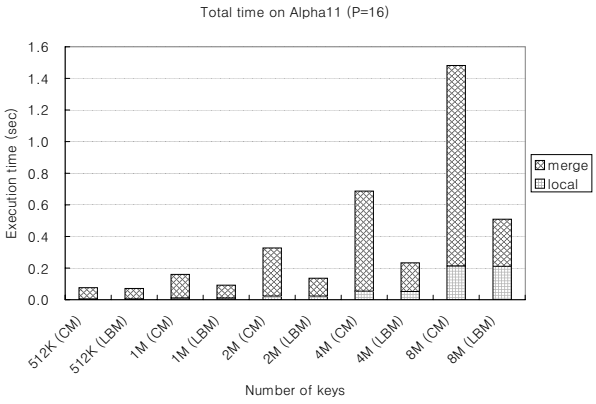


(c) PC cluster

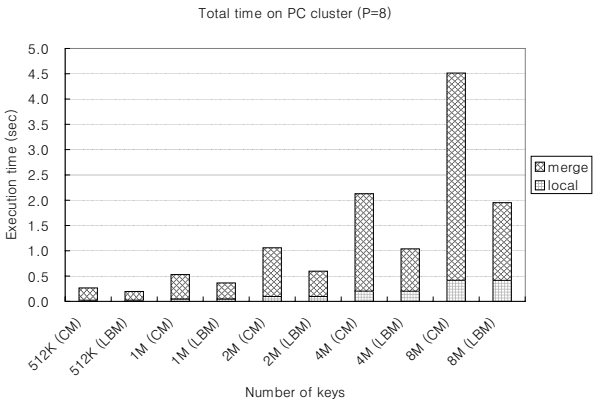
Figure 4. Speedups of merge time on three machines



(a) T3E

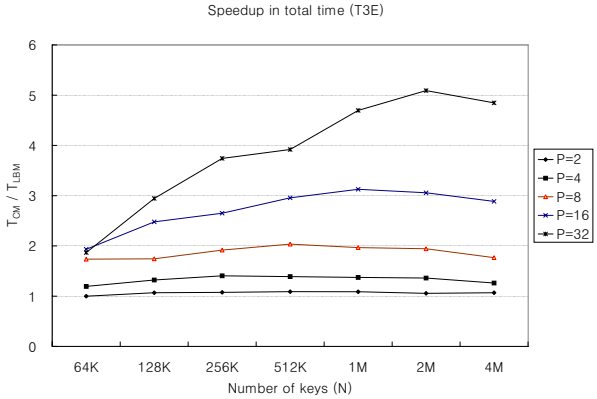


(b) Alpha11

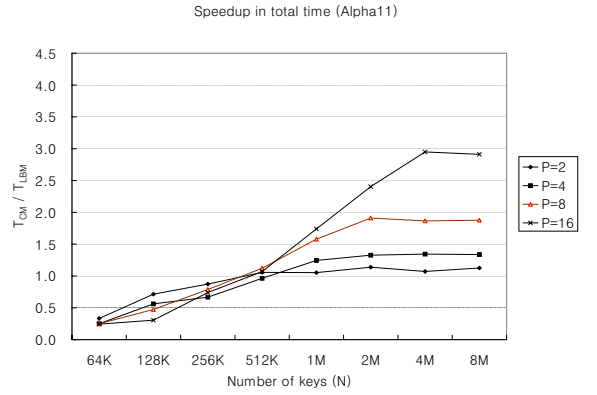


(c) PC cluster

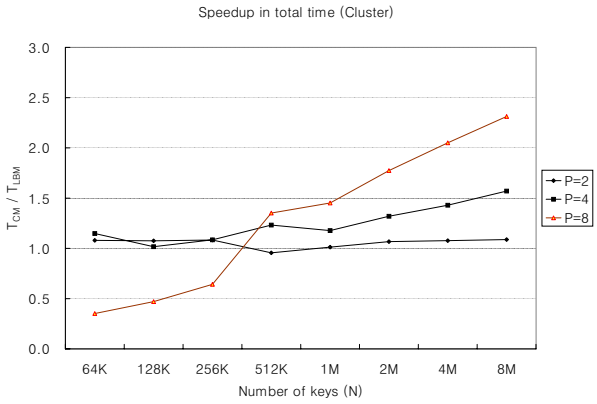
Figure 5. Total sorting time on three machines



(a) T3E



(b) Alpha11



(c) PC cluster

Figure 6. Speedups of total sorting time on three machines