International Journal of Parallel Programming, Vol. 31, No. 1, February 2003 (© 2003)

Parallel Merge Sort with Load Balancing

Minsoo Jeon[1] and Dongseung Kim[1]

Received August, 2002; revised October, 2002

Parallel merge sort is useful for sorting a large quantity of data progressively. The merge sort should be parallelized carefully since the conventional algorithm has poor performance due to the successive reduction of the number of participating processors by half, and down to one in the last merging stage. The proposed load-balanced merge sort utilizes all processors throughout the computation. It evenly distributes data to all processors in each stage. Thus every processor is forced to work in all phases. Significant performance enhancement has been achieved up to a speedup of $(P-1)/\log P$ where P is the number of processors. Experimental results demonstrate a speedup of 9.6 (upper bound of 10.7) on 32-processor Cray T3E when sorting 4M 32-bit integers, and a speed up of 2.3 (upper bound of 2.8) on an 8-node PC cluster.

KEY WORDS: Merge sort; parallel algorithm; load balancing; splitter.

1. INTRODUCTION

Many comparison-based sequential sorts take $O(N \log N)$ time to sort N keys. To speedup the sorting, multiprocessors are employed for parallel sorting. Several parallel sorting algorithms such as bitonic sort,[1,2] sample sort,[3] column sort,[4] and parallel radix sort[5,6] have been devised. Parallel sorts usually need a fixed number of data exchange and merging operations. The computation time decreases as the number of processors grows. Since sorting time is dependent on the size of the data set each processor has to compute, good load balancing is important. In addition, if interprocessor communication is not fast such as occurs in distributed memory

[1] Department of Electrical Engineering, Korea University, Seoul, 136-701, Korea. E-mail: {msjeon,dkim}@classic.korea.ac.kr

computers, the amount of overall data to be exchanged and the frequency of communication have a great impact on the total execution time.

Merge sort is frequently used in many applications. Parallel merge sort using the PRAM model was reported to have a faster execution time of $O(\log N)$ for N input keys using N processors.(7) However, distributed-memory based parallel merge sort is slow because it needs a local sort followed by a fixed number of merge iterations, which includes lengthy communication. The major drawback of the conventional parallel merge sort is the fact that load balancing and processor utilization get worse as it iterates; in the beginning every processor participates in merging of the list of N/P keys with their 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 the merging process. This results in the low utilization of resources. Consequently, it lengthens the computation time. This paper introduces a new parallel merge sort scheme, called *load-balanced parallel merge sort*, that forces every processor to participate in merging at every iteration. Each processor deals with a list of size of about N/P at every iteration, thus the load of processors is kept balanced to reduce the execution time.

The paper is organized as follows. In Section 2 we present the conventional and improved parallel merge sort algorithms together with an explanation of how more parallelism is obtained. Section 3 reports experimental results performed on a Cray T3E and a PC cluster. We conclude in the last section followed by performance analysis in the appendix.

2. PARALLEL MERGE SORT

2.1. Simple Method

Parallel merge sort goes through two phases: a local sort phase and a merge phase. The local sort phase produces keys in each processor sorted locally. Then in the merging phase, processors merge them in $\log P$ steps as explained below. In the first step, processors are paired (sender, receiver). Each sender sends its list of N/P keys to its partner (the receiver), then the two lists are merged by each receiver to make a sorted list of $2^1 N/P$ keys. Half of the processors work during the merge, and the other half sit idling. In the next step only the receivers 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 $2^2 N/P$ keys. The process continues until a complete sort list of N keys is obtained (Fig. 1). The detailed algorithm is given in Algorithm 1.

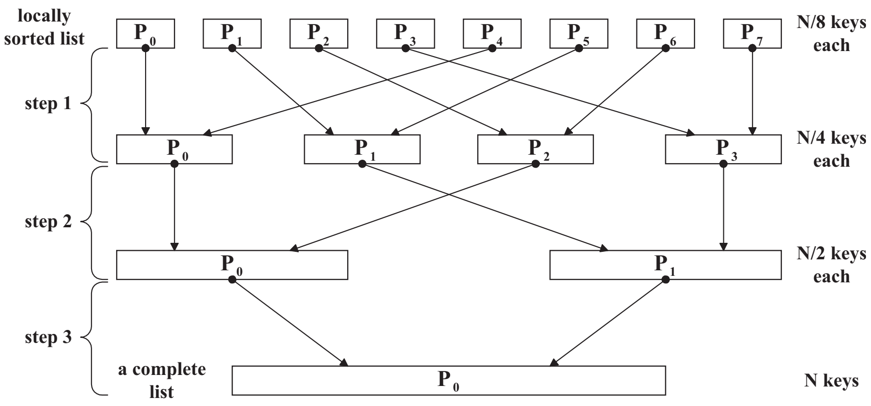


Fig. 1. Conventional parallel merge sort with 8 processors.

As mentioned earlier, the algorithm does not fully utilize all processors. A simple calculation reveals that only $P/\log P$ $(=(P/2+P/4+P/8+\cdots+1)/(\log P \text{ steps}))$ processors out of P processors are used on average. Therefore, it must have inferior performance to an algorithm that makes a full use of the processors.

Algorithm 1. Simple 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 $0 \leqslant i \leqslant P-1$
       $P_i$ sorts a list of $N/P$ keys locally.
  2. for $j = 0$ to $(\log P) - 1$ do
       forall $0 \leqslant i \leqslant h-1$
         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
```

2.2. Load-Balanced Parallel Merge Sort

To keep each list of sorted data in one processor is relatively simple. However, as the size of the lists grows, sending them to other processors for merging is time consuming, and processors that no longer keep lists after transmission sit idling until the end of the sort. The key idea in our parallel sort is to distribute each (partially) sorted list onto multiple processors such that each processor stores an approximately equal number of keys, and all processors take part in merging throughout the execution. Figure 2 illustrates this idea for a merge with 8 processors, where each rectangle represents a list of sorted keys, and processors are shown in the order that they store and merge the corresponding list. It would invoke more parallelism, and thus shorten the sort time. One difficulty in this method is to find a way to merge two lists, each of which is distributed in multiple processors, rather than store them on a single processor. Our design for minimizing key movement is 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. It also computes a *histogram* of its own keys. The histogram plays an important role in determining the minimum number of keys to be exchanged during the merge. Processors keep nondecreasing (or nonincreasing) order for their keys. In the first merging step, all groups have a size of one processor, and each group is paired with another group called its *partner group*. In this step, there is only one communication partner per processor. Each pair exchanges its two boundary keys (a minimum and a maximum keys) and determines the new order of the two processors

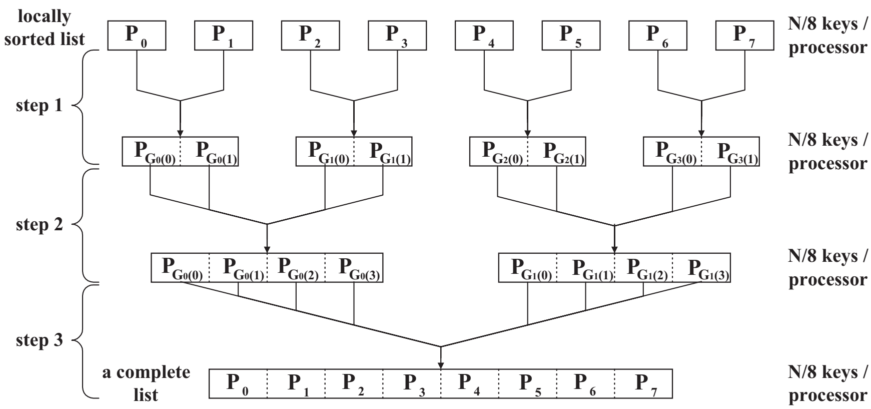


Fig. 2. Load-balanced parallel merge sort.

according to the minimum key values. Now each pair exchanges group histograms and computes a new one that covers both. Each processor then divides the intervals (histogram bins) of the merged histogram into two parts (i.e., *bisection*) so that the (half) lower indexed processor will keep the smaller half of the keys, and the higher will keep the larger half. Now each processor sends out the keys that will belong to other processor(s) (for example, those keys in the shaded intervals are transmitted to the other processor in Fig. 3). Each processor merges the keys with those arriving from its paired processor. Now each processor holds $N/P \pm \Delta$ keys because the bisection of the histogram bins may not be perfect (we hope Δ is relatively small compared to N/P). The larger the number of histogram bins, the better the load balancing. In this process, only the keys in the overlapped intervals need to merge. It implies that keys in the non-overlapped interval(s) do not interleave with keys of the partner processor's during the merge. They are simply placed in a proper position in the merged list. Often there may be no overlapped intervals at all, and therefore no keys are exchanged.

From the second step and on, the group size (i.e., the number of processors per group) doubles. The merging process is the same as before except that each processor may have multiple communication partners up to the group size in the worst case. Now boundary values and group histograms are again exchanged between paired groups, then the order of processors is decided and the histogram bins are divided into 2^i parts (at the ith iteration). Keys are exchanged between partners, then each processor merges the received keys with its own. One cost saving method is used here called *index swapping*. When a processor has to send most of its keys to its partner and also receive the equal amount from it, we rather swap the logical ids of the two, instead of moving a large amount of keys among them. Thus, index swapping minimizes the number of key exchange. The procedure of the parallel sort is summarized in Algorithm 2.

Algorithm 2. Load-balanced parallel merge sort

```
1. Each processor sorts a list of $N/P$ keys locally and obtains local
   histogram.
2. Iterate log $P$ times the following computation
    2.1. Each group of processors exchanges boundary values
         between its partner group and determines the logical ids of
         processors for the merged list.
```

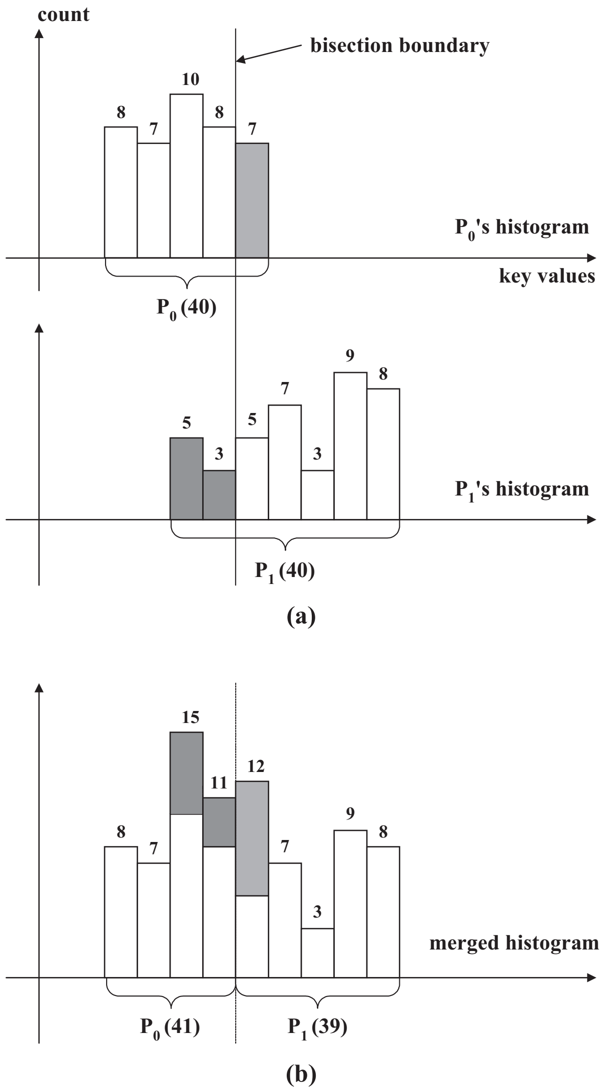


Fig. 3. Example of exchanging and merging keys by using histogram at first iteration (Processors P_0,P_1 both have 40 keys before merge, and 41 & 39 after merge, respectively.) [(a) before merge, (b) after merge].

2.2. Each group exchanges histograms with its paired group and computes a new histogram, then divides the bins into 2^{i-1} equal parts.
/* At ith iteration, there are $P/2^{i-1}$ groups, each of which includes 2^{i-1} processors */

2.3. Each processor sends keys to the designated processors that will belong to others due to the division.

2.4. Each processor locally merges its keys with the received ones to obtain a new sorted list.

2.5. Broadcast logical ids of processors for the next iterations.

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 and histogram data at each step, and a broadcast of 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.

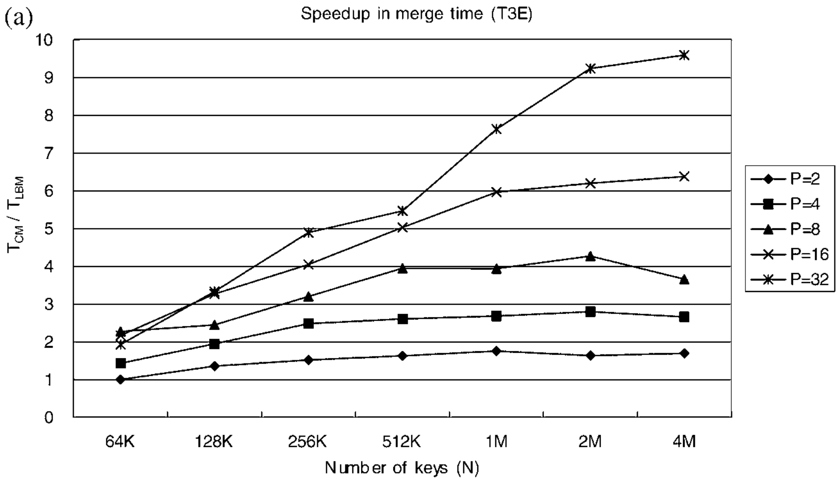
3. EXPERIMENTAL RESULTS

The new parallel merge sort has been implemented on two different parallel machines: a Cray T3E and a Pentium III PC cluster. The T3E consists of 450 MHz Alpha 21164 processors and a 3-D torus network. Pentium III PC cluster is a set of 8 PCs with 1 GHz Athlon CPUs interconnected by a 100 Mbps Fast Ethernet switch. The maximum number of keys is limited by the capacity of the main memory of each machine. Keys are synthetically generated with two distribution functions (*uniform* and *gaussian*) with 32-bit integers for PC cluster, and 64-bit integers for T3E. Code is written in the C language with the MPI communication library.

Table I. Comparison of Predicted and Measured Speedups on T3E and PC Cluster with 4M Keys

	P	2	4	8	16	32
T3E	$\eta_{\text{predicted}}$	1.73	2.58	4.02	6.46	10.68
	η_{measured}	1.69	2.66	3.66	6.38	9.60
PC cluster	$\eta_{\text{predicted}}$	1.18	1.78	2.76	–	–
	η_{measured}	1.14	1.60	2.30	–	–

The parameters of the computation and communication performance of individual systems were measured. K_1 and K_2 are the average time to transmit one key and the average time per key to merge N keys, respectively. For the T3E, K_1 and K_2 are 0.048 and 0.125 msec/key respectively, and C is calculated to be 1.732 according to Eq. (9) in the Appendix. For the PC cluster, K_1 and K_2 are 0.386 and 0.083 msec/key, and C is 1.184.



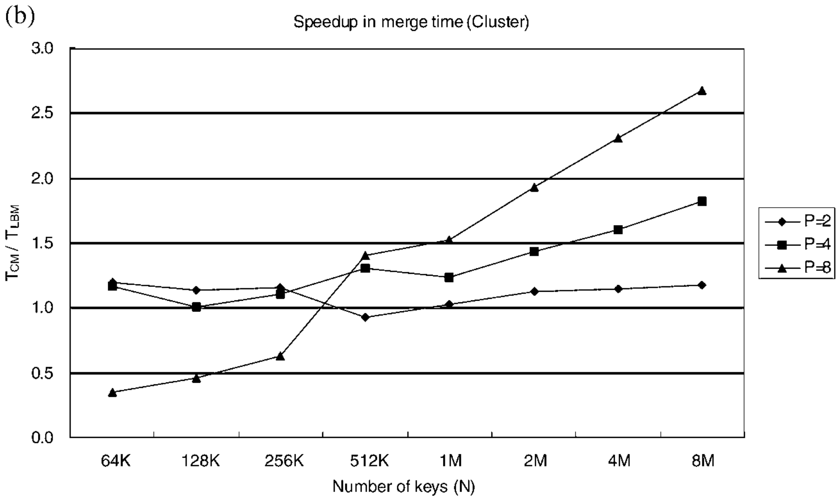
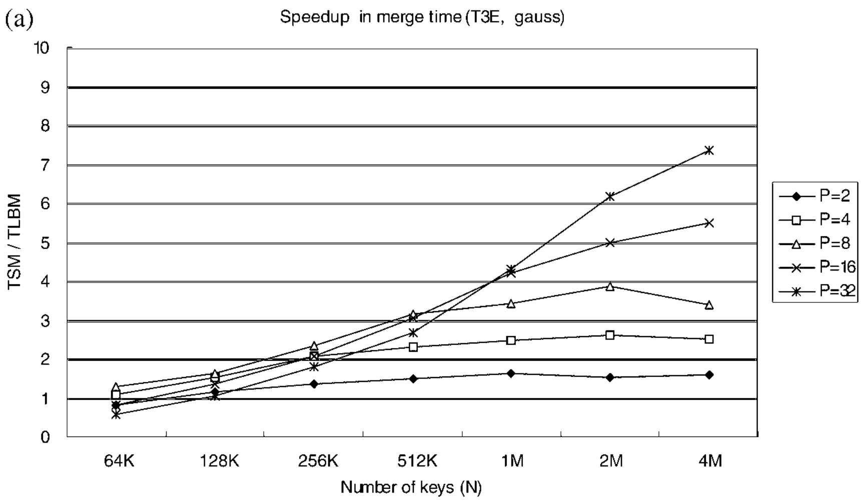


Fig. 4. Speedups of merge time on two machines with uniform distribution [(a) T3E, (b) PC cluster].

Notice that T3E is expected to achieve the greater performance enhancement due to having the bigger C introduced in Eq. (9) in the Appendix. The predicted and measured speedups of the T3E and the PC cluster are recorded in Table I. Most of the results are close to the predicted ones. The speedups *in merge time only* of the load-balanced merge sort over the



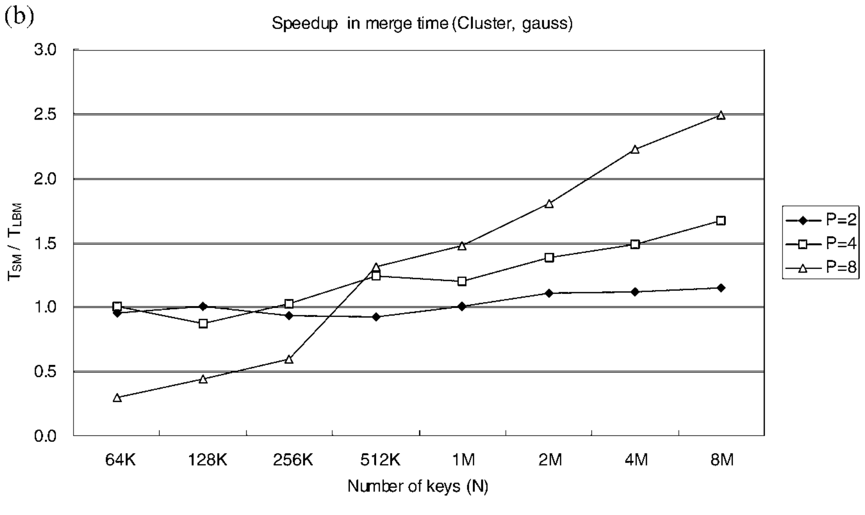
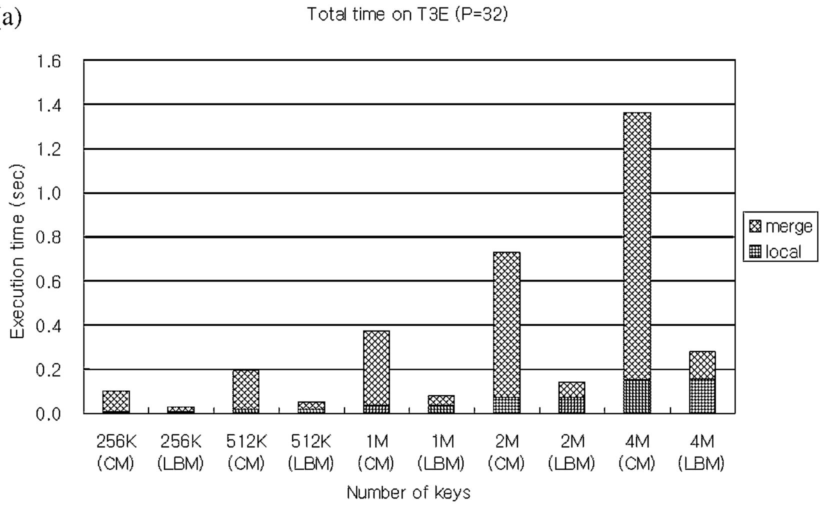


Fig. 5. Speedups of merge time on two machines with gaussian distribution [(a) T3E, (b) PC cluster].

conventional merge sort are shown in Figs. 4 and 5. The speedups with gaussian distribution are smaller than those with uniform distribution since Δ in Eq. (7) is larger in the gaussian distribution than in the uniform distribution. The improvement gets better 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 relatively large

(a)



(b)

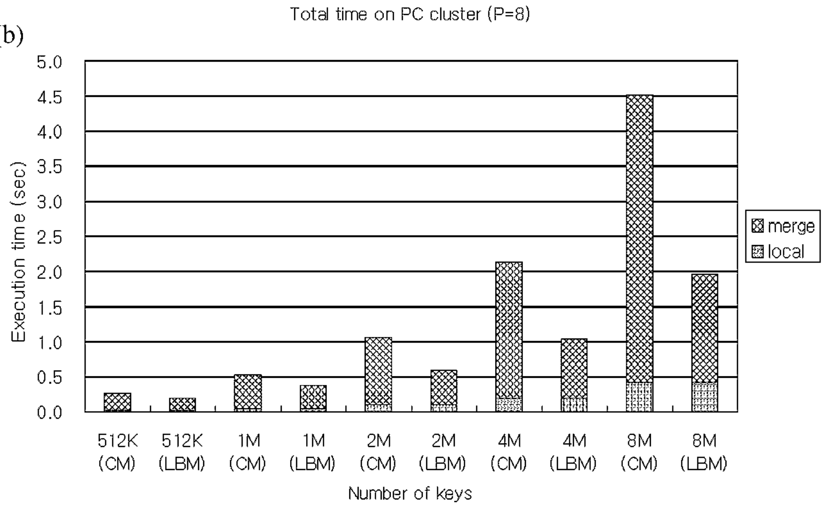


Fig. 6. Both the local (sequential) sort time and the parallel merge time are shown for the two machines with uniform distribution [(a) T3E, (b) PC cluster].

overhead in exchanging boundary values and histogram information, and broadcasting processor ids. Experimental results on the T3E demonstrates higher speedup, which matches the analytic result given in Eq. (8). The comparisons of *the total sorting time* and distribution of the load balanced merge sort with the conventional algorithm are shown in Fig. 6. The execution time for the merging phase is significantly shortened, whereas the local sort time for both methods remains the same as on one machine. The best speedups of 9.6 and 2.3 for the merging phase are achieved on the Cray T3E with 32 processors and an 8-node PC cluster respectively.

4. CONCLUSION

We have improved parallel merge sort by distributing and computing approximately equal number of keys in all processors throughout the merging phases. Using the histogram information, keys can be divided equally regardless of their distribution. We have achieved a maximal speedup of 9.6 when merging 4M keys on a 32-processor Cray T3E, which is about 90% of the upper bound. We have also reached a maximal speedup of 2.3 for 4M keys on an 8-node PC cluster, which is about 83% of the upper bound. This scheme can be applied to a parallel implementation of similar merging algorithms such as parallel quick sort.

APPENDIX

The upper bound of the speedup of the new parallel merge sort will now be estimated. Let $T_{\mathrm{seq}}(N/P)$ be the time for the initial local sort to make a sorted list. $T_{\mathrm{comp}}(N)$ represents the time for merging two lists, each with $N/2$ keys, and $T_{\mathrm{comm}}(M)$ is the interprocessor communication time to transmit M keys. For the input of N keys, $T_{\mathrm{comm}}(M)$ and $T_{\mathrm{comp}}(M)$ are estimated as follows:[8]

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

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

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 Algorithm 1, step 1 requires $T_{\mathrm{seq}}(N/P)$. Step 2 repeats log P times, so execution time of the simple parallel merge sort (SM) is estimated as below:

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

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 $T_{\text{seq}}(N/P)$. The time required in steps 2.1 and 2.2 is ignorable if the number of histogram bins is small compared to N/P. Since the maximum number of keys assigned to each processor is N/P, at most N/P keys are exchanged between paired processors in step 2.3. Each processor merges $N/P+\Delta$ keys in step 2.4. Step 2.5 requires $O(\log P)$ time. The communication of steps 2.1 and 2.2 can be ignored since the time is relatively small compared to the communication time in step 2.3 if N/P is large (coarse grained). Since step 2 is repeated $\log P$ times, the execution time of the load-balanced parallel merge sort (LBM) can be estimated as below:

$$T_{\text{LBM}}(N,P)=T_{\text{seq}}\left(\frac{N}{P}\right)+\log P\cdot\left\{T_{\text{comm}}\left(\frac{N}{P}\right)+T_{\text{comp}}\left(\frac{N}{P}+\Delta\right)\right\} \tag{4}$$

To observe the enhancement in the *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:

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

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

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

$$\eta=\frac{T_{\text{SM}}(N,P)}{T_{\text{LBM}}(N,P)}=\frac{K_1\cdot\frac{N}{P}(P-1)+K_2\cdot\frac{2N}{P}(P-1)}{K_1\cdot\frac{N}{P}\log P+K_2\cdot(\frac{N}{P}+\Delta)\log P} \tag{7}$$

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:

$$\eta = \frac{K_1 \cdot \frac{N}{P}(P-1) + K_2 \cdot \frac{2N}{P}(P-1)}{K_1 \cdot \frac{N}{P}\log P + K_2 \cdot \frac{N}{P}\log P} = \frac{K_1 + 2K_2}{K_1 + K_2} \cdot \frac{P-1}{\log P}$$
$$= C \cdot \frac{P-1}{\log P} \tag{8}$$

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

$$C = \frac{K_1 + 2K_2}{K_1 + K_2} = 1 + \frac{K_2}{K_1 + K_2} = 1 + \frac{1}{K_1/K_2 + 1} \tag{9}$$

ACKNOWLEDGMENTS

This research was supported by KOSEF Grant (No. R01-2001-00341).

REFERENCES

1. K. Batcher, Sorting Networks and Their Applications, *Proceedings of the AFIPS Spring Joint Computer Conference 32*, Reston, VA, pp. 307–314 (1968).
2. Y. Kim, M. Jeon, D. Kim, and A. Sohn, Communication-Efficient Bitonic Sort on a Distributed Memory Parallel Computer, *International Conference on Parallel and Distributed Systems* (*ICPADS'2001*) (June 2001).
3. J. S. Huang and Y. C. Chow, Parallel Sorting and Data Partitioning by Sampling, *Proceedings of 7th Computer Software and Applications Conference*, pp. 627–631 (November 1983).
4. A. C. Dusseau, D. E. Culler, K. E. Schauser, and R. P. Martin, Fast Parallel Sorting under Log *P*: Experience with the CM-5, *IEEE Transactions on Computers*, Vol. 7 (August 1996).
5. S. J. Lee, M. Jeon, D. Kim, and A. Sohn, Partitioned Parallel Radix Sort, *J. Parallel Distr. Comput.* (*JPDC*), **62**:656–668 (2002), also in *3rd International Symposium on High Performance Computing* (*ISHPC'2000*), Tokyo, Japan, pp. 160–171 (October 2000).
6. A. Sohn and Yuetsu Kodama, Load Balanced Parallel Radix Sort, *Proceedings of the 12th ACM International Conference on Supercomputing* (July 1998).
7. R. Cole, Parallel Merge Sort, *SIAM J. Comput.*, **17**(4):770–785 (1998).
8. R. Hockney, Performance Parameters and Benchmarking of Supercomputers, *Parallel Computing*, **17**(10/11):1111–1130 (December 1991).