JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 12, 171–177 (1991)

Merging Multiple Lists on Hierarchical-Memory Multiprocessors*

PETER J. VARMAN AND SCOTT D. SCHEUFLER

Department of Electrical and Computer Engineering, Rice University, Houston, Texas 77251-1892

BALAKRISHNA R. IYER

Data Base Technology Institute, IBM Programming Systems, P.O. Box 49023, San Jose, California 95161-9023

AND

GARY R. RICARD

IBM Corporation, D511 015-3, Hwy. 52 and 37th Street NW, Rochester, Minnesota 55901

An efficient multiprocessor algorithm to merge m, $m \geq 2$, sorted lists containing N elements is described. In a hierarchical-memory multiprocessor, the proposed method substantially reduces the data access costs in comparison with traditional schemes that successively merge the lists two at a time. The proposed method uses a novel partitioning algorithm to split up the m lists into *range-disjoint* parts of *equal* size. Each of the parts can then be merged concurrently and independently by a different processor using an *m-way* merge. The time complexity to find a partition is $O(m \log(fN/m)$, where f, $0 < f < 1$, is a fraction that determines the size of a partition. Implementation results for multiway merging using the partitioning algorithm on a Sequent Symmetry multiprocessor are described.

1. INTRODUCTION

The problem of merging multiple sorted lists arises naturally in several important applications like database management systems. Multilist merging is a basic procedure used for sorting data on computer systems with a hierarchical memory structure. For instance, external sorting methods (i.e., those dealing with the situation where the data is too large to fit in primary storage) typically work by first creating a number of sorted runs on secondary storage and then merging as many runs as possible using *k-way* merges [1]. Similar multilist merging schemes have been proposed for parallel systems [2–5] and for distributed systems [6], with the aim of reducing the communication complexity of sorting.

* Part of this research was performed while the first author was visiting the IBM Almaden Research Center, San Jose. Part was supported by NSF Grant CCR-9006300.

A number of cost-optimal parallel algorithms for merging *two* sorted lists can be found in the recent literature [7–14]. To merge m, $m \geq 2$, lists using this method, the lists are repeatedly merged, two at a time, until a single sorted list is obtained. Such a scheme can lead to an optimal cost algorithm in terms of the number of comparisons performed. However, this strategy is inefficient whenever data accesses are expensive compared to typical instruction-execution times. Such a situation arises, for instance, in external sorting, where it is essential to minimize the number of times a data element is accessed from secondary storage. A similar situation arises in multiprocessor systems with private processor caches and shared main memory, since exclusive access to data in the private cache is significantly faster than access to slower shared memory. Merging the lists two at a time requires $\log_2 m$ passes over the data, where in each pass data are moved between the levels of the memory hierarchy. In contrast, an *m-way* merge of the lists requires only one pass over the data, resulting in a substantial reduction in the cost of the algorithm.

In this paper we describe an efficient parallel algorithm to merge m, $m \geq 2$, sorted lists containing N elements, using p processors. The scheme uses a novel partitioning algorithm to break up the m lists into p *range-disjoint* parts of *equal* sizes. That is, the partitioning algorithm allows one to efficiently find the smallest N/p elements of the data, the $2N/p$ smallest elements, etc. Each processor can then independently merge the elements in its range of data. Concatenating the output of the individual merges results in the desired sorted output.

The main results reported in the paper are as follows. In Section 2, we describe and analyze the partitioning algorithm. The time complexity to find a partition consisting of the smallest $\lceil fN \rceil$ elements of m lists with N data elements is $O(m \log(fN/m)$, where f, $0 < f < 1$, is a fraction that

0743-7315/91 $3.00

determines the size of the partition. In Section 3, we describe and report experimental results for multiway merging using the partitioning algorithm, on the Sequent Symmetry multiprocessor [15]. Speedups of 15 using 16 processors and roughly 128K elements were measured. The paper concludes with Section 4.

1.1. *Review of Previous Work*

For an extensive discussion on *sequential* multilist merging algorithms and their application in sorting, the reader is referred to [1]. A number of papers in the literature on parallel merge–sort deal with the merging of *two* sorted lists [7–14]. In particular, [9–14] show how to partition two lists into equal-sized, range-disjoint parts; however, there is no direct generalization of these schemes to multiple lists. In contrast, there has been much less work reported on multilist merging for multiprocessors [2–5]. The schemes in [2–4] concurrently merge sets of sorted runs until the number of runs equals the number of processors; these runs are then merged sequentially by one processor. The best time possible in such a scheme is $\Omega(N)$ since there is at least one sequential pass through the data. The method in [5] partitions the runs into range-disjoint parts of *approximately* equal sizes, using a simple partitioning algorithm. However, the sizes of the individual partitions may be substantially different, particularly if the distribution of data values is skewed. The algorithm does not guarantee load balance for the final merge. In contrast, the method described in this paper parallelizes the final merge step and ensures load balance, since the partitioning algorithm can break up the elements of m sorted runs into an arbitrary number of range-disjoint parts, with an *equal number* of elements in each part.

Previously, the problem of selecting the kth smallest element of a set of m sorted sequences had been studied in the context of combinatorial optimization, and an asymptotically optimal algorithm for this problem was provided by Fredrickson and Johnson [16]. Somewhat surprisingly, the application of such selection algorithms to partition data for merging seems to have been largely ignored, except in the context of distributed systems (for example, see [6]). Two cases are considered in [16]: those where $k \leq m$ and $k > m$. The first case is solved by an elegant $O(m)$ procedure which we use as a substep in our algorithm. Note however that for small m, due to the large constants involved in this procedure, it may be advisable to solve this by a simpler strategy using a "tournament tree" [1]. In this method, the smallest element is determined in m comparisons and the subsequent smallest elements in an additional $\log m$ comparisons each, for a complexity of $O(m + (k - 1)\log m)$ which is $O(m \log m)$.

The latter case, $k \geq m$, is the one that arises in the partitioning of the lists for merging. It is solved in [16] by an iterative method, where each iteration discards roughly 1/10th of the data, using a "weighted selection" procedure. Our algorithm is much simpler and more intuitive and is, in a sense, a mirror image of this method. In each iteration we consider larger and larger samples of the input, until the entire data set is considered. Our algorithm converges more quickly since the size of the input considered in each iteration is (more than) doubled; in [16], the size reduces by roughly a factor of 9/10 in each iteration.

2. PARTITIONING ALGORITHM

2.1. *Preliminaries*

Let $A^1, \ldots, A^m$ be m lists sorted, without loss of generality, in increasing order. Let N denote the total number of elements in the combined lists and $0 < f < 1$, an arbitrary fraction. The goal of the partitioning algorithm is to identify the $\lceil fN \rceil$ smallest elements in the lists.

DEFINITION. Let S be a set of elements with an implied total ordering among the elements. An *f-partition* of S is a partition of S into two disjoint subsets L and H that satisfy the following two conditions:

Domination criterion: H dominates L, i.e., $\forall h \in H$ and $\forall l \in L$, $h > l$, and

Size criterion: $|L| = \lceil f|S| \rceil$.

We sketch below the algorithm to efficiently find the *f-partition* of the set consisting of all the elements in A^i, $\forall i = 1, \ldots, m$. A formal description and a complexity analysis are provided in the next section. The algorithm is iterative. In each iteration a suitable *sample* of the elements of the lists is considered. Let $S(i)$ denote the set of elements considered at the ith iteration. Assume inductively that an *f-partition* of $S(i)$ into two disjoint subsets, $L(i)$ and $H(i)$, has been found. For the $(i + 1)$th iteration, the sample, $S(i + 1)$, consists of (roughly) double the number of elements in $S(i)$; an *f-partition* of $S(i + 1)$ into $L(i + 1)$ and $H(i + 1)$ is determined. At the last iteration, the sample will consist of all the N elements. Determining an *f-partition* of this sample results in a solution to the problem. The crux of the algorithm is that by suitably choosing the samples, one can compute the *f-partition* of $S(i + 1)$ from the *f-partition* of $S(i)$ with only a small amount ($O(m)$) of computational overhead. Since $|S(i + 1)| > 2|S(i)|$, the number of iterations grows no more than logarithmically with the number of elements.

2.2. *Formal Description*

Let $A^i = a(i, 1), a(i, 2), \ldots, a(i, n)$, $1 \leq i \leq m$, be the ith ordered sequence assumed without loss of generality to be sorted in increasing order. Let f be a fraction, $0 < f < 1$. The problem is to determine an *f-partition* of $S = \{a(i, j), 1 \leq i \leq m, 1 \leq j \leq n\}$.

Assume that $n = 2^r - 1$, for some integer $r \geq 0$. In Section 2.3 we indicate how to handle the case when this assumption

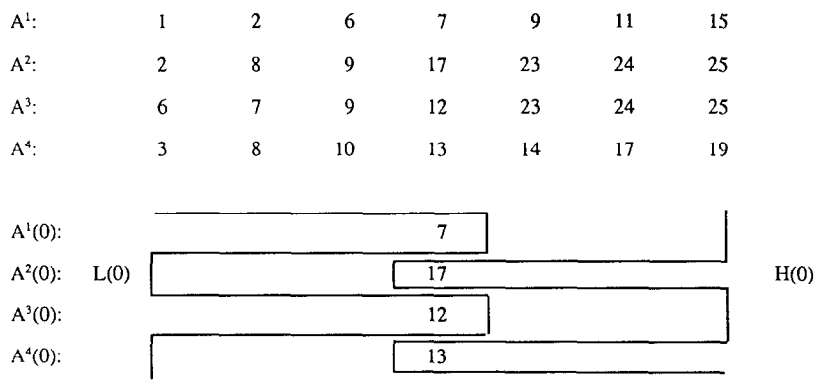

FIG. 1a. Partitioned samples after iteration $k = 0$.

is not true, including the case when the lists are of unequal sizes. In that section we also describe how to handle the base case of the algorithm.

Let $w_k = 2^{r-k-1}$, and let $A^i(k)$, $0 \leqslant k \leqslant r-1$, denote the subsequence of A^i consisting of every w_kth element of A^i. That is, $A^i(k) = a(i, w_k), a(i, 2w_k), a(i, 3w_k), \ldots, a(i, (2^{k+1}-1)w_k)$. Let $S(k)$ denote the set of elements belonging to $A^i(k)$, $\forall i$, $1 \leqslant i \leqslant m$.

DEFINITIONS. Let $L(k)$, $H(k)$ be an *f-partition* of $S(k)$.

(i) The *boundary* of $A^i(k)$ (denoted b_i), $1 \leqslant i \leqslant m$, is given by

$$b_i = \begin{cases} 0 & \text{if } L(k) \cap A^i(k) = \Phi \\ \max\{j : a(i,j) \in L(k) \cap A^i(k)\} & \text{otherwise.} \end{cases}$$

(ii) The largest element of $L(k)$, denoted *lmax*, is given by $lmax = \max\{a(i, b_i), i = 1, \ldots, m\}$.

Figure 1 illustrates the definitions and operation of the algorithm for the case $m = 4$, $f = \frac{1}{2}$, and $n = 7$. The basis for the induction consists of the sample $S(0)$ shown in Fig. 1a. $S(0)$ consists of one element (the fourth) from each list. The *f-partition* of $S(0)$ is shown in the figure. As may be seen, $L(0)$ consists of the elements 7 and 12 and $H(0)$ the elements 13 and 17. The boundary indices at this iteration are given by $b_1 = 4$, $b_2 = 0$, $b_3 = 4$, and $b_4 = 0$. Also $lmax = 12$.

Figure 1b shows the sample $S(1)$. This consists of every second element from each sequence. We now illustrate the inductive step of the algorithm. For each sequence, $A^i(1)$, there is exactly one element of $S(1)$ that does not lie within the boundaries of the partition established in the previous iteration; this is the element of $S(1)$ that immediately follows the boundary element of $A^i(0)$. These elements (11, 8, 24, and 8) are shown in boldface in Fig. 1b. We refer to these elements as *undecided* elements; the remaining elements of $S(1)$ are called *decided* elements.

The decided elements of $S(1)$ are naturally partitioned into two disjoint subsets $\hat{L}$ and $\hat{H}$. $\hat{L}$ consists of the elements $a(i,j)$ of $S(1)$ such that $j \leqslant b_i$, and $\hat{H}$ consists of the decided elements $a(i, j)$ of $S(1)$ such that $j > b_i$. In the example, $\hat{L}$ consists of the elements 7, 2, 12, and 7 and $\hat{H}$ of the elements 17, 24, 13, and 17. Note that $\hat{H}$ dominates $\hat{L}$. (See Fig. 1b).

Now we compare each undecided element with *lmax*, the largest element in $L(0)$. If an undecided element is smaller than *lmax* it is moved into $\hat{L}$; otherwise it is moved into $\hat{H}$. In the example, three of the four undecided elements (8, 8, and 11) are smaller than $lmax = 12$, while the remaining undecided element (24) is greater than *lmax*. Figure 1c shows partitions $\hat{H}$ and $\hat{L}$ after these undecided elements have been moved. Note that we now have a partition of $S(1)$ into $\hat{H}$ and $\hat{L}$ that satisfies the domination criterion. Also note that this required exactly m comparisons (m is the number of lists being partitioned).

To reestablish the size criterion, a certain number of elements may have to be moved between the partitions. If $|\hat{L}| = \lceil f|S(1)|\rceil$, we have an *f-partition* of $S(1)$, with $L(1) = \hat{L}$ and $H(1) = \hat{H}$. If not, then if $|\hat{L}| > \lceil f|S(1)|\rceil$, we must move the $(|\hat{L}| - \lceil f|S(1)|\rceil)$ *largest* elements of $\hat{L}$ to

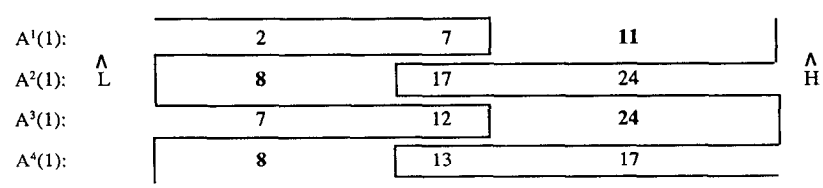

FIG. 1b. Samples for iteration $k = 1$.

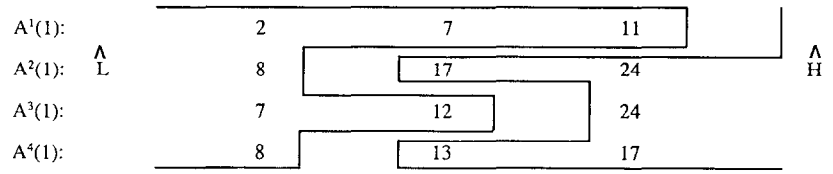

FIG. 1c. Reestablishing the dominance condition.

$\hat{H}$; otherwise we must move the $(\lceil f|S(1)|\rceil - |\hat{L}|)$ *smallest* elements of $\hat{H}$ to $\hat{L}$. We show in Lemma 1 that in any case the number of elements to be moved is no more than m. Also the elements to be moved can be identified in $O(m)$ time.

Returning to the example, we see that $|S(1)| = 12$, $\lceil f|S(1)|\rceil = 6$, and $|\hat{L}| = 7$. Hence, one element has to be moved from $\hat{L}$ to $\hat{H}$. The largest element of $\hat{L}$ (i.e., 12) is moved. Figure 1d shows the *f-partition* of $S(1)$, obtained by moving this element from $\hat{L}$ to $\hat{H}$. Finally Fig. 1e shows the *f-partition* of $S(2)$, by repeating the steps outlined above for the next iteration. This solves the original problem.

A formal description of the inductive step of the algorithm follows.

Inductive Step. Given $S(k)$, $L(k)$, and $H(k)$, determine $L(k+1)$ and $H(k+1)$. Assume that the boundary indices b_i and *lmax* for $S(k)$ have been computed prior to the start of iteration $k+1$.

Step 1.
Let $U = \{u_i : u_i = a(i, b_i + w_{k+1}),\ i = 1, \ldots, m\}$ /* set of *undecided* elements */
Let $\hat{L} = \{a(i,j) \in S(k+1),\ j \leq b_i,\ i = 1, \ldots, m\}$. /* small valued *decided* elements */
Let $\hat{H} = S(k+1) - U - \hat{L}$. /* *large valued decided* elements */
For each $u_i \in U$ do: *if* $u_i < lmax$, Add u_i to $\hat{L}$; *else*: Add u_i to $\hat{H}$.

Step 2.
$goal = \lceil f|S(k+1)|\rceil$. /* *required size of* $L(k+1)$. */
if $|\hat{L}| > goal$:
Move the $(|\hat{L}| - goal)$ *largest elements of* $\hat{L}$ *to* $\hat{H}$
else if $|\hat{L}| < goal$
Move the $(goal - |\hat{L}|)$ *smallest elements of* $\hat{H}$ *to* $\hat{L}$
$L(k+1) = \hat{L}$; $H(k+1) = \hat{H}$
Update b_i, $i = 1, \ldots, m$, and *lmax*.

LEMMA 1. *The number of elements to be moved from* $\hat{L}$ *to* $\hat{H}$ *(or vice versa) in Step* 2 *of the algorithm is no more than* m.

Proof. At the start of Step 1, we have by definition

$$|S(k+1)| = 2|S(k)| + m,$$
$$\hat{L} = 2|L(k)|, \quad \hat{H} = 2|H(k)|, \quad |U| = m.$$

At most $m-1$ elements of U can be smaller than *lmax*. Hence, at most $m-1$ elements can be added to $\hat{L}$ in Step 1. Consequently, at the end of Step 1, the size of $\hat{L}$ is bounded by $2|L(k)| \leq \hat{L} \leq 2|L(k)| + m - 1$.

In Step 2, consider first the case when $|\hat{L}| > goal$. In this case the number of elements to be moved is given by $|\hat{L}| - goal \leq 2|L(k)| + m - 1 - \lceil f|S(k+1)|\rceil$.

Since $S(k)$ was correctly *f-partitioned*, $|L(k)| = \lceil f|S(k)|\rceil$. Also, $|S(k+1)| = 2|S(k)| + m$. Now, $\lceil 2\lceil f|S(k)|\rceil - \lceil f(2|S(k)| + m)\rceil\rceil \leq 2\lceil f|S(k)|\rceil - \lceil 2f|S(k)|\rceil - \lceil fm\rceil + 1$. Hence, $|\hat{L}| - goal \leq m - \lceil fm\rceil + 2\lceil f|S(k)|\rceil - \lceil 2f|S(k)|\rceil \leq m - \lceil fm\rceil + 1 \leq m$.

Similarly, if $goal > \hat{L}$, then $goal - \hat{L} \leq \lceil 2f|S(k)| + fm\rceil - 2\lceil f|S(k)|\rceil \leq \lceil fm\rceil \leq m$.

The implementation of Step 2 allows several alternatives. Note from Lemma 1 that this step requires identifying the k smallest (or largest) of the elements of m sorted lists, where $k \leq m$. Hence, this can be done easily using the standard tournament tree [1] based method in $O(m \log m)$ time, or by the method of Fredrickson and Johnson [16] in $O(m)$ time (refer to the discussion in Section 1.1).

2.3. *Base Case and Complexity Analysis*

Given m lists, each sorted in increasing order, and a fraction $\hat{f}$, $0 < \hat{f} < 1$, let the total number of elements in the m sorted lists be N. We describe here the base step of the algorithm for finding $\lceil \hat{f}N\rceil$ smallest elements of the lists.

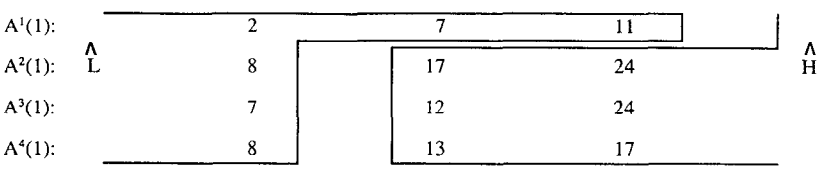

FIG. 1d. Partitioned samples after iteration $k = 1$.

$A^1(2)$:		1	2	6	7	9	11	15	
	∧								∧
$A^2(2)$:	L	2	8	9	17	23	24	25	H
$A^3(2)$:		6	7	9	12	23	24	25	
$A^4(2)$:		3	8	10	13	14	17	19	

FIG. 1e. Final partition after iteration $k = 2$.

To simplify the discussion we make the following assumptions about the values of $\hat{f}$, N, and m. These cover the majority of cases that occur in practice. We also indicate how to handle cases not covered by the assumptions.

(i) $\hat{f}N > m$, (ii) $\hat{f} \leq \frac{1}{2}$, and (iii) $m \geq 2$.

In the case when assumption (i) is not satisfied, the problem reduces to finding the smallest k elements of the given lists where $k \leq m$; this can be directly solved as indicated in Section 1.1. The case when $\hat{f} > \frac{1}{2}$ (assumption (ii)) can be treated by finding the $(1 - \hat{f})$-*partition* of lists sorted in *decreasing* order and handled symmetrically. Assumption (iii) is self-explanatory.

There are two main steps: The first is a preprocessing step to account for the fact that the lists may be of unequal size or of length different from one less than an integer power of two, as assumed for the inductive step described previously. The second step chooses a small number of these elements and determines a partition of them. This serves as the basis of the induction.

Step 1 (Preprocessing). Define r to be the integer such that $2^{r-1} < \hat{f}N/m \leq 2^r$. From assumption (i), $r \geq 1$. Let $n_{\max}$ denote the number of elements in the longest of the m lists, and define $\alpha = \lfloor n_{\max}/2^r \rfloor$. Conceptually, pad the end of each list with $+\infty$, so that the length of each list is n, where $n = 2^r(\alpha + 1) - 1$. Let A^i, $i = 1, \ldots, m$, denote these lists. The total number of elements in these augmented lists is mn. The given problem of finding the smallest $\lceil \hat{f}N \rceil$ elements of the original lists is equivalent to finding the *f-partition* of the elements in A^i, $i = 1, \ldots, m$, where $f = \lceil \hat{f}N \rceil / nm$.

Step 2 (Choosing a sample for the base case). For the base case of the partitioning algorithm, choose the 2^rth element of each A^i. That is, $A^i(0) = \{a(i, 2^r), a(i, 2 \cdot 2^r), a(i, 3 \cdot 2^r), \ldots, a(i, \alpha \cdot 2^r)\}$, $i = 1, \ldots, m$. Find an *f-partition* of these αm elements as discussed below. This *f-partition* of $S(0)$ serves as the basis for the inductive steps of the algorithm.

Determining an *f-partition* of $S(0)$ in Step 2 above requires finding its smallest $\lceil f\alpha m \rceil$ elements. Lemma 2 shows that this requires identifying no more than the $2m$ smallest elements of the lists $A^i(0)$, $i = 1, \ldots, m$. Hence, this can be accomplished either directly in $O(m \log m)$ time using the tournament tree strategy [1] or in $O(m)$ time by two applications of the scheme of [16]. Lemma 3 shows that the base case is well defined; i.e., in the definition above, $\alpha \geq 1$ and hence $S(0)$ is not empty. Following this basis step, there will be r iterative steps before the algorithm terminates.

LEMMA 2. $\lceil f\alpha m \rceil \leq m$.

Proof. $\alpha = \lfloor n_{\max}/2^r \rfloor$, $1/2^r \leq m/\hat{f}N$, $n \geq n_{\max}$, and $f = \lceil \hat{f}N \rceil / nm$. By direct use of these relations, $\alpha \leq \lfloor \lceil \hat{f}N \rceil / f\hat{f}N \rfloor$. Therefore, $f\alpha \leq 1 + \lfloor \lceil \hat{f}N \rceil / \hat{f}N \rfloor \leq 2$, for $\hat{f} \geq 1/N$. Hence, $\lceil f\alpha m \rceil \leq m$.

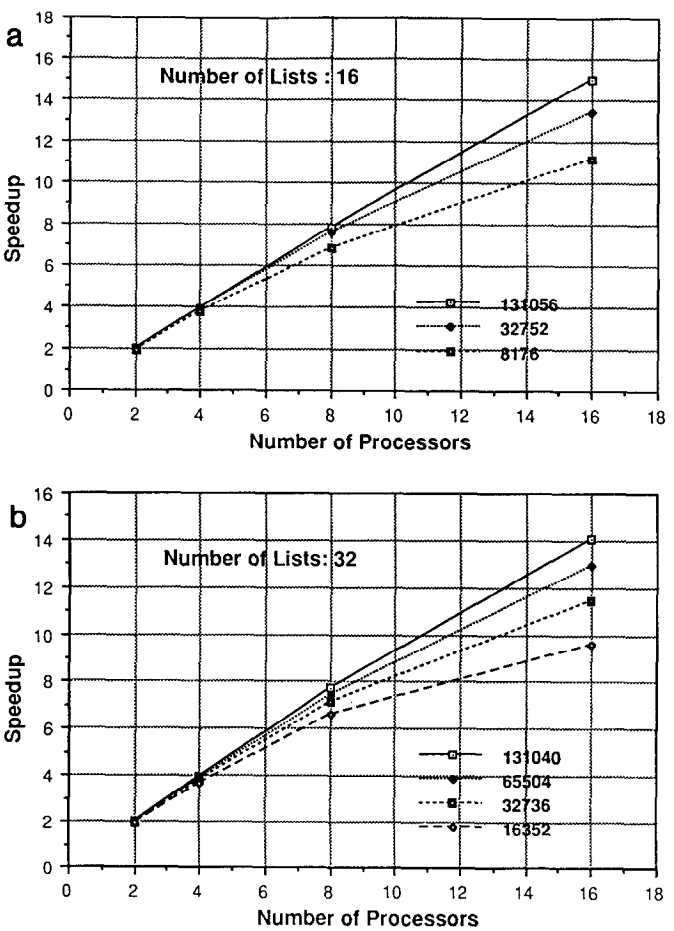

FIG. 2. (a) Speedups with 16 lists. (b) Speedups with 32 lists.

LEMMA 3. $\alpha \geq 1$.

Proof. By definition, $\hat{f}N/m > 2^{r-1}$. By assumption (ii), $\hat{f}2^r \leq 2^{r-1}$. Hence, $N > m2^r$, and consequently, $n_{\max} > 2^r$. Thus, $\alpha = \lfloor n_{\max}/2^r \rfloor \geq 1$.

THEOREM. *The f-partition of the elements in m sorted lists having a total of N elements can be found in* $O[m + m \log(fN/m)]$ *time.*

Proof. The base case of the algorithm takes $O(m)$ time, as discussed above. As discussed in Section 2.2, each iteration of the algorithm can be done in $O(m)$ time. The number of iterations is r, where $r < 1 + \log(fN/m)$. Hence the theorem.

3. IMPLEMENTATION RESULTS

A parallel multilist merging algorithm based on the above partitioning strategy was implemented on a Sequent Symmetry multiprocessor with 20 processors and its performance measured. A brief description of the program is as follows.

(A) Allocate space and create sorted arrays of random data.
(B) Create dummy processes.
(C) Fork( ).
(D) Initialize data structures.
(E) Partition (Concurrent execution by $p - 1$ processors).
(F) Merge (Concurrent execution by p processors).
(G) Join( ).

Time was measured using the microsecond clock from before Step C until after Step G. This block was repeated thrice for each set of data, and the time used was obtained by discarding the largest and smallest running time obtained. The same code was used for a one-processor merge; however, the time required for executing the code in Steps C, E, and G in this case was negligible. Speedups were computed with respect to the time required for a one-processor merge.

Processor 1 does not take part in the partitioning (Step E), but begins merging (Step F) from the start of the array, stopping when it has merged N/p elements. Processor i, $2 \leq i \leq p$, computes the start of the ith partition and then begins merging elements from that point, until it has merged N/p elements. Consequently no synchronization is required between Steps E and F. The merging Step F used a standard k-way tournament tree based algorithm [1]. The data for the program consisted of randomly generated 4-byte integers.

Figure 2 shows the measured speedups. Figures 2a and 2b show the speedups when the elements are distributed among 16 and 32 sorted lists, respectively. Using 16 processors, for a data size of approximately 128K elements, the speedup is 15 when a 16-way partitioning and merge is performed and 14 when a 32-way partitioning and merge is performed.

4. CONCLUSIONS

We have presented a parallel algorithm for merging m sorted lists, $m \geq 2$. The importance of such a multilist merging algorithm arises whenever data accesses are expensive, as is the case in a system with hierarchical memory. The method is based on an efficient partitioning algorithm that allows the m lists to be split into an arbitrary number of equal-sized parts that can be independently merged by different processors in parallel. Thus, load balance in the merge can be achieved at a small cost for the partitioning algorithm. The complexity of the partitioning algorithm is $O[m \log(fN/m)]$, to find an *f-partition* of N elements distributed among m lists. This time is optimal [16].

The parallel merging algorithm was implemented on a Sequent Symmetry multiprocessor, and the speedups measured empirically. For a data size of 128K elements, a speedup of 15 using 16 lists and 16 processors was obtained. The speedup is expected to improve as the size of the data gets larger. Work on determining the best sorting strategy on different hierarchical-memory multiprocessor architectures is continuing.

REFERENCES

1. Knuth, D. *The Art of Computer Programming. Vol. 3. Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.
2. Valduriez, P., and Gardarin, G. Join and semijoin algorithms for multiprocessor database machines *ACM Trans. Data Base Systems* **9,** 1 (1984), 133–161.
3. Beck, M., Bitton, D., and Wilkinson, W. K. Sorting large files on a backend multiprocessor. *IEEE Trans. Comput.* **C-37,** 7 (1988).
4. Gray, J., Stewart, M., Tsukerman, A., Uren, S., and Vaughen, B. Fastsort: An external sort using parallel processing. *Tandem System Rev.* **2** (Dec. 1986).
5. Quinn, M. J. Parallel sorting algorithms for tightly coupled multiprocessors. *Parallel Comput.* **6,** 3 (1988), 349–357.
6. Rotem, D., Santoro, N., and Sidney, J. B. Distributed sorting. *IEEE Trans. Comput.* **C-34,** 4 (1985), 372–375.
7. Batcher, K. E. Sorting networks and their applications. *Proc. AFIPS 1968 Spring Joint Computer Conference,* 1968, pp. 307–314.
8. Cole, R. Parallel merge sort. *SIAM J. Comput.* **17,** 4 (1988), 770–785.
9. Akl, S. G., and Santoro, N. Optimal parallel merging and sorting without memory conflicts. *IEEE Trans. Comput.* **C-36,** 11 (1987), 1367–1369.
10. Bilardi, G., and Nicolau, A. Bitonic sort with $O(n \log n)$ comparisons. *Proc. 20th Annual Conference on Information Science and Systems,* 1986.
11. Frances, R. S., and Mathieson, I. D. A benchmark parallel sort for shared memory multiprocessors. *IEEE Trans. Comput.* **C-37,** 12 (1988), 1619–1626.
12. Deo, N., and Sarkar, D. Optimal parallel algorithms for merging and sorting. *Proc. 3rd International Conference on Supercomputing,* 1988, pp. 513–521.
13. Varman, P., Iyer, B., Haderle, D., and Dunn, S. Parallel merging: Algorithm and implementation results. *Parallel Comput.* **15** (1990), 165–177.

14. Wheat, M. A parallel sorting algorithm for tightly-coupled multiprocessors. In *Concurrency: Practice and Experience.* Wiley, New York, Mar. 1990, Vol. 2, No. 1, pp. 27–32.

15. Lovett, T., and Thakkar, S. The symmetry multiprocessor system. *Proc. 1988 International Conference on Parallel Processing.* IEEE Computer Society, Silver Spring, MD, 1988, pp. 303–310.

16. Fredrickson, G. N., and Johnson, D. B. The complexity of selection and ranking in $X + Y$ and matrices with sorted columns. *J. Comput. System Sci.* **24** (1982), 197–208.

PETER J. VARMAN received his B.Tech from the Indian Institute of Technology, Kanpur, and the M.S. and Ph.D. degrees from the University of Texas at Austin. Since 1983 he has been with the Department of Electrical and Computer Engineering at Rice University, where he is currently an associate professor. He has been a visiting faculty member at IBM T.J. Watson and Almaden Research Centers. His research interests include multiprocessor algorithms and architectures. VLSI, and fault tolerance.

BALAKRISHNA (BALA) R. IYER received his B.Tech. from the Indian Institute of Technology, Bombay, and the M.S. and Ph.D. degrees from Rice University, Houston. He has worked for Bell Telephone Laboratories, Murray Hill, New Jersey, and has been with IBM since 1982. His research interests include parallel processing, data base management, and data compression.

GARY R. RICARD received his B.S. degrees in chemistry and computer science from the South Dakota School of Mines and Technology in 1980 and 1981. At IBM he has worked in the S/38 database area, and continues work on IBM's AS/400 as an advisory programmer. His major interests are in operating systems, computer graphics, and fractals.

SCOTT D. SCHEUFLER received his B.S. and M.E.E. degrees in electrical engineering from Rice University. He is currently with Texas Instruments at Houston.

Received August 1990; revised January 1991; accepted January 20, 1991