Accepted for Publication in *IEEE Transactions on Computers*

Time-Space Optimal Parallel Merging and Sorting*

Xiaojun Guan† and **Michael A. Langston**‡

Abstract. A parallel algorithm is *time-space optimal* if it achieves optimal speedup and if it uses only a constant amount of extra space per processor even when the number of processors is fixed. Previously published parallel merging and sorting algorithms fail to meet at least one of these criteria. In this paper, we present a parallel merging algorithm that, on an EREW PRAM with k processors, merges two sorted lists of total length n in $O(n/k + \log n)$ time and $O(k)$ extra space, and is thus time-space optimal for any value of $k \leq n/(\log n)$. We also describe a *stable* version of our parallel merging algorithm that is similarly time-space optimal on an EREW PRAM. These two parallel merges naturally lead to time-space optimal parallel sorting algorithms.

Key Words. block rearranging, internal buffering, memory management, merging and sorting, parallel computation, time-space optimality

---

* A preliminary version of a portion of this paper was presented at the International Conference on Parallel Processing, held in St. Charles, Illinois, in August, 1989.

† Department of Computer Science, Washington State University, Pullman, WA 99164–1210. This author's research has been supported in part by the Washington State University Graduate Research Assistantship Program.

‡ Department of Computer Science, University of Tennessee, Knoxville, TN 37996–1301, and Department of Computer Science, Washington State University, Pullman, WA 99164–1210. This author's research has been supported in part by the National Science Foundation under grants MIP–8603879 and MIP–8919312, and by the Office of Naval Research under contract N00014–88–K–0343.

1. Introduction

The quest for efficient parallel merging and sorting algorithms has been a long-standing topic of intense interest, as evidenced by the impressive volume of literature published on this subject (see, for example, [Ak, BDHM, LDM] for recent surveys). Much of the focus has been on the search for methods that are *optimal* in the classic sense that asymptotically optimal speedup is attained[1]. Indeed, a number of parallel algorithms have been proposed that are optimal under this criterion, including those found in [AKS, AS, BH, BS, Co, Kr, SV, Va].

Curiously, and quite unlike the case for sequential algorithms, very little attention seems to have been paid to space management issues. Some of this phenomenon can perhaps be attributed to the fact that much of what is known about parallel algorithms is relatively new. Accordingly, less time has elapsed for practical problems of implementation to become widely known. (See, for example, the formidable difficulties in memory management that have recently been encountered when an attempt has been made to implement $\mathcal{NC}$-style algorithms[2] on hypercubes with 16 and 256 nodes [BB].) Another contributing factor may be that memory has become so inexpensive during the last few years that it is often easy simply to ignore it. In any event, space utilization continues to be a critical aspect in many applications, even for sequential processing; this criticality is only heightened in parallel

[1]A parallel method attains asymptotically optimal speedup if the product of the number of processors it employs and the amount of time it takes is within a constant factor of the time required by a fastest sequential algorithm.

[2]A problem is said to be in $\mathcal{NC}$ if it possesses a parallel algorithm that, for any problem instance of size n, employs a number of processors bounded by some polynomial function of n and requires an amount of time bounded by some polylogarithmic function of n.

processing systems when the number of processors is bounded.

None of the previously published parallel merging and sorting strategies are *time-space optimal*. That is, none achieve optimal speedup and, at the same time, require only a constant amount of extra space per processor even when the number of processors is fixed. We remark that, from a consideration of time alone, these algorithms represent an acceptable approach, mirroring one reason for the popularity of the parallel random-access machine (PRAM) model. Specifically, if the number of processors is fixed, then as the problem size grows, an $\mathcal{NC}$ algorithm can be "scaled down," so that each real processor needs merely to emulate multiple virtual processors, thereby accounting for the massive parallelism inherent in the design of the algorithm. Unfortunately, however, space requirements in this scenario tend to "blow up," unless the extra space required by each real processor is constant, independent of the growing problem size. Added cause for concern is that, even if enough global memory is available, the more shared memory accesses a program makes the more message traffic is placed on whatever interconnection network is used to realize the shared memory, with an attendant downgrading of the overall system's performance. Incorporating secondary memory devices into this picture naturally leads to additional problems [RB], to be avoided as long as main memory need not be squandered on temporary extra storage.

From the foregoing discussion, we conclude that any genuine attempt to minimize extra space dictates that the total number of extra storage cells required by each processor be constant, even when the number of processors available is bounded by some constant, k, whose value is independent from the size of a problem instance, n. (This is in contrast to work such as that described in [RS], in which constant extra space is employed at each processor, but the number of processors is assumed to be $\Theta(n^2)$.) Moreover, as an attractive side effect of attempting to minimize extra space, a bounded number of processors reflects more faithfully any real parallel computing environment.

In this paper, we present for the first time a parallel merging algorithm that, on an exclusive-read exclusive-write (EREW) PRAM, merges two sorted lists in $O(n/k + \log n)$ time and constant extra space per processor, and hence is time-space optimal for any value of $k \leq n/(\log n)$. We also describe how this gives rise to a stable[3] version of our parallel merging algorithm that is similarly time-space optimal on an EREW PRAM. We observe that (our technique for achieving) stability incurs two penalties: a slightly more complicated algorithm and somewhat larger constants of proportionality. These two parallel merges naturally lead to time-space optimal parallel sorting algorithms.

In the next section, we briefly review the main features of a recently-published linear-time in-place sequential merge. Although a direct parallelization of this method is not possible, its overall structure is helpful in simplifying the presentation of our parallel algorithm, which we describe in detail in Section 3. As we demonstrate in that section, a major factor in our algorithm's asymptotic time-space optimality is the introduction of a useful technique that is based on what we dub a *displacement table*. We next move on to the subject of stable merging, describing in Section 4 how some relatively simple modifications to our parallel merge can be exploited to yield a stable time-space optimal parallel algorithm. Extensions

[3] A merging algorithm is stable if it preserves the original relative order of records with identical keys.

to sorting and open topics for future research are discussed in the final section.

2. A Review of Time-Space Optimal Sequential Merging

To simplify the presentation of our time-space optimal parallel algorithm in the next section, it is useful first to review at least briefly the recently-published and relatively simple linear-time, in-place sequential merge from [HL1]. Expectedly, some of the operations that are easy to perform sequentially are difficult to perform in parallel. Interestingly, on the other hand, some of the operations that are difficult to perform sequentially are easy to perform in parallel. On the whole, however, it turns out that a direct parallelization of this novel sequential method is not possible. Nevertheless, its overall structure can be used to *guide our thinking* so that, with the aid of our parallel displacement table technique to be presented later, we can direct all available processors to work efficiently in unison.

The (sequential) optimality attained with respect to both time and space inherently relies on the related notions of *block rearranging* and *internal buffering*. To get a feel for the general way in which such a strategy works, it is helpful to view a list containing n records as a collection of $\Theta(\sqrt{n})$ blocks, each of size $\Theta(\sqrt{n})$. This approach allows us to employ one block as the (internal) buffer to aid in resequencing the other blocks of the two sorted sublists and then merging these blocks into one sorted list. Since only the contents of the buffer and the relative order of the blocks need ever be out of sequence, linear time is sufficient to achieve order by straight-selection sorting [Kn] both the buffer and the blocks (each sort involves $O(\sqrt{n})$ keys). We refer the interested reader to [HL1–HL4] for extensive background, related results and additional details on block rearranging and internal buffering methods.

We note that, for the sake of complete generality, we allow neither the key nor any other part of a record to be modified by our algorithms. Such is necessary, for example, when records are write-protected or when there is no explicit key field within each record, but instead a record's key is a function of one or more of its data fields.

Let L denote a list containing two sublists to be merged, each with its keys in nondecreasing order. We shall make a few simplifying assumptions about L to facilitate the discussion. (See [HL1] for a complete exposition of the algorithm, an example, and the $O(\sqrt{n})$ time and $O(1)$ space implementation details necessary for handling arbitrary inputs.)

We assume that n is a perfect square, and that the records of L have already been permuted so that $\sqrt{n}$ largest-keyed records are at the front of the list (their relative order there is immaterial), followed by the remainders of the two sublists, each of which we now assume contains an integral multiple of $\sqrt{n}$ records in nondecreasing order. Therefore, we can view L as a series of $\sqrt{n}$ blocks, each of size $\sqrt{n}$. The leading block will be used as an internal buffer to aid in the merge.

The first step is to sort the $\sqrt{n} - 1$ rightmost blocks by their *tails* (rightmost elements), after which their tails form a nondecreasing key sequence. (In this setting, selection sort requires only $O(n)$ key comparisons and record exchanges.) Records within a block retain their original relative order.

The second step, which is the most complex, is to direct a sequence of series merges. An initial pair of series of records to be merged is located as follows. The first series begins with the head of block 2 and terminates with the tail of block i, $i \geq 2$, where block i is the first block such that the key of the tail of block i exceeds the key of the head of block $i+1$. The second series consists solely of the records of block $i+1$. The buffer is used to merge these two series. That is, the leftmost unmerged record in the first series is repeatedly compared to the leftmost unmerged record in the second, with the smaller-keyed record swapped with the leftmost buffer element. Ties are broken in favor of the leftmost series. (In general, the buffer may be broken into two pieces as the merge progresses.) This task is halted when the tail of block i has been moved to its final position.

The next two series of records to be merged are now located. This time, the first begins with the leftmost unmerged record of block $i+1$ and terminates as before for some $j \geq i$. The second consists solely of the records of block $j+1$. The merge is resumed until the tail of block j has been moved. This process of locating series of records and merging them is continued until a point is reached at which only one such series exists, which is merely shifted left, leaving the buffer in the last block.

The final step is to sort the buffer, thereby completing the merge of L.

$O(n)$ time suffices for this entire procedure, because each step requires at most linear time. $O(1)$ space suffices as well, since the buffer was internal to the list, and since only a handful of additional pointers and counters are necessary.

3. Time-Space Optimal Parallel Merging on the EREW PRAM Model

Note that, exclusive of implementation details for extracting the internal buffer and for handling lists and sublists of arbitrary sizes, the sequential algorithm just described comprises three steps: *block sorting, series merging* and *buffer sorting*. Unfortunately, these steps do not appear to permit a direct parallelization, at least not one that requires only constant extra space per processor. In particular, the internal buffer is instrumental in the series merging step, dictating a block size of $\Theta(\sqrt{n})$ that in turn severely limits what can be accomplished efficiently in parallel.

Optimistically, however, observe that if we could only devise a time-space optimal method to:

(1) use bigger blocks (namely, one block for each of the k processors, giving rise to a block size of n/k, a value that might be unboundedly greater than $\sqrt{n}$) and

(2) reorganize the file so that the problem is reduced to one of k local merges (that is, replace the contents of each block with two sublists, one from each of the two original sublists in L, so that the largest key in block i is no greater than the smallest key in block $i+1$, for $1 \leq i < k$),

then we could complete a time-space optimal merge of L by simply directing each processor to merge the contents of its own block using the algorithm sketched in the last section. This observation is the genesis of the parallel method we shall now present.

Ignoring for the moment implementation details for dealing with lists and sublists of arbitrary sizes (these details will be addressed at the end of this section), our parallel method comprises these five steps: *block sorting, series delimiting, displacement computing, series splitting* and *local merging*. Since the last step of our algorithm (local merging) is easy from a parallel standpoint, it is not surprising that the earlier steps are relatively complicated, requiring a careful coordination of all processors to achieve efficiently the desired reorganization of the file.

To facilitate discussion, let us temporarily assume that the number of records in each of the two sublists in L is evenly divisible by k. We shall refer to a record or block from the first sublist of L as an $L1$ record or an $L1$ block. We shall use the terms $L2$ record and $L2$ block in an analogous fashion for elements from the second sublist.

Block Sorting. We first view L as a sequence of k blocks, each of size n/k. See Figure 1, in which we employ a handy pictorial representation for L, using the vertical axis to indicate increasing key values and the horizontal axis to indicate increasing record indices.

INSERT FIGURE 1

We seek to sort these blocks by their tails. This is a relatively simple chore if one is willing to settle for a concurrent-read exclusive-write (CREW) algorithm. For example, we could begin by directing each $L1$ ($L2$) processor to perform a binary search on the $L2$ ($L1$) sublist, comparing its block's tail against the tails in that sublist. In order to sort the blocks efficiently on the EREW model, we adopt a slightly more complex strategy.

We first direct each processor to set aside a copy of the tail of its block and its index (an integer between 1 and k, inclusive). We can now merge the k tail copies (dragging along the indices) by reversing in parallel the copies from the second sublist and then invoking the well-known bitonic merge [Ba], a task requiring $O(\log k)$ time and $O(k)$ total extra space.

After this merge is completed, each processor knows the index of the block it is to receive. With the use of but one extra storage cell per processor, it is now a simple matter for the processors to acquire their respective new blocks in parallel without memory conflicts, one record at a time (say, from the first record in a block to the last). This task requires $O(n/k)$ time and $O(k)$ extra space.

This completes the block sorting step, and has required $O(n/k + \log k)$ time and constant extra space per processor. See Figure 2.

INSERT FIGURE 2

Series Delimiting. As with the sequential method, it is helpful at this point to think of the list as containing a collection of pairs of series of records, with each pair of series to be merged. (Of course, we cannot now merely mimic the sequential series merging step. If there are large series, then it would take too long to merge them; if there are large blocks, then it would take too long to sort any type of internal buffer.) We require a somewhat more refined definition of "series," however, because we must insist that pairs of series do not overlap one another. The first and second series of any given pair meet as before, where the tail of block i exceeds the head of block $i+1$. To determine where pairs meet each other, we now use the term "breaker" to denote the first record of block $i+1$ that is no smaller than the tail of block i. Thus the first series of a pair needs only to begin with a breaker, and the second series of that pair needs only to end with the record immediately preceding the next breaker. This notion is illustrated in Figure 3. Because each pair of series is made up either of a portion of an $L1$ block followed by zero or more full $L1$ blocks and a portion of an $L2$ block, or a portion of an $L2$ block followed by zero or more full $L2$ blocks and a portion of an $L1$ block, and because these two configurations are symmetric, we shall henceforth address only the former case in this and subsequent figures.

INSERT FIGURE 3

For a processor to determine whether its block contains a second series, it simply compares its head to its left neighbor's tail. If this comparison reveals that the processor does contain such a series, then it invokes a binary search to locate its breaker (it must have one — recall that the blocks were first sorted by their tails) and broadcasts[4] the breaker's location first to its left and then to its right. By this means, a processor learns the location of the breaker to its immediate right and the location of the breaker to its immediate left.

From this it follows that every processor can correctly delimit the one or two pairs of series that are relevant to the contents of its block in $O(\log(n/k) + \log k)$ time and constant extra space per processor.

Displacement Computing. Recall that our goal is to reorganize the file so that local merging is possible as a final step. This requires an efficient parallel means for splitting each pair of series among the processors that are in charge of the pair's blocks. In order to accomplish this, we shall now introduce what we term a *displacement table*, with one table

[4]A convenient algorithm for this type of broadcasting can for example be found in [Ul, page 234], where it is termed a "data distribution algorithm." Alternately, such broadcasting can be efficiently accomplished with parallel prefix computation.

entry to be stored at each processor. In this table we seek to enumerate, for each processor with a block (or portion thereof) from the first series, the number of records from the second series that would displace records in that block *if* there were no other records in the first series. Thus a displacement table is of immediate use in the next step (series splitting), because processor i needs only to know its entry, E_i, and the entry for processor $i-1$, E_{i-1}. From these two values it is easy for processor i to determine the number of its records that are to be displaced by records from the left (namely, E_{i-1}) and the number that are to be displaced by records from the second series (namely, $E_i - E_{i-1}$).

INSERT FIGURE 4

As with the block sorting step, things are relatively simple if one is willing to settle for a CREW algorithm. For example, we could begin by directing each processor whose block contains records from the first series to perform a binary search on the second series. In order to compute the displacement table entries efficiently on the EREW model, we adopt a considerably more complicated strategy. In particular, we must solve a nontrivial *processor allocation problem* [SV]. We agree with the sentiment expressed by others (see, for example, [BH, KR]) that details relevant to this thorny subject warrant a careful exposition.

For an arbitrary pair of series, let f denote the index of the processor handling the first record in the first series, and let p denote the number of blocks with records in that series. Thus processor $f+p$ is responsible for the second series. We seek to direct the p processors with records in the first series to work in unison and without memory conflicts to determine where each of their block's tails would need to go if they were merged with the $m < n/k$ records of the second series. To accomplish this, we now present a technique that is perhaps best described as a sequence of phases of operations.

In the first phase, each processor with records in the first series sets aside a copy of its block's tail and its index (an integer between f and $f + p - 1$, inclusive). Each also sets aside two pieces of information from the second series: processor i ($f \leq i < f + p$) computes and saves a copy of the offset $h = (i - f + 1)(m/p)$ and a copy of the hth record of the second series. We can now merge the $2p$ elements made up of p tails and p selected records (dragging along the indices and the offsets) by reversing in parallel the selected records and then invoking a bitonic merge, a task requiring $O(\log p)$ time and $O(p)$ extra space.

After this, each processor with records in the first series examines the two keys in its temporary storage. If a processor finds a tail, then (with the use of the tail's index) it reports its own index to the processor handling the block from which the tail originated. Thus every processor can determine from the movement of its block's tail just how many of the records selected from the second series are smaller, and therefore which of the p subseries of the second series, each subseries of size m/p, to merge into next. In order for a processor to be able to determine how many other tails are to be merged into the same next subseries as its block's tail, each one compares its next subseries with that of its neighbors. If the comparison reveals a subseries boundary, then broadcasting is used to inform the other processors of the location of this boundary (as we did when broadcasting a breaker's location in the series delimiting step).

For the second and each subsequent phase, processors proceed as in the first phase, but now with new offsets and selected records based on the proper subseries into which their block's tails are to be merged and the number of other tails that are also to be merged there. Processors continue to iterate this procedure until each has determined where its block's tail would go if it were merged with the other tails and the second series. Note that some processors may be employed in as few as $\log_k m$ phases, each requiring $O(\log k)$ time, while others may simultaneously be employed in as many as $\log_2 m$ phases, each requiring constant time. In general, letting the sequence $k_1, k_2, ..., k_l$ denote the number of tails in any chain of recursive calls, we observe that $k_1 \times k_2 \times ... \times k_l$ is $O(m)$, and hence $\log k_1 + \log k_2 + ... + \log k_l$ is $O(\log m)$. Therefore, $O(\log n)$ time and $O(k)$ extra space has been consumed up to this point.

Let l_i $(1 \leq l_i \leq m + p)$ denote the location that the tail of the block of processor i $(f \leq i < f + p)$ would occupy in a sublist containing the p tails and the entire second series if such a sublist were available. Processor i now computes $l'_i = l_i - (i - f) - 1$, to eliminate the effect of its block's tail and all preceding tails. It next employs two pointers to compare a record in its block, beginning at location n/k (its tail), to a record in the second series, beginning at location l'_i, repeatedly decrementing the pointer that points to the larger key for l'_i iterations. (We insist that each processor works from right to left in its interval of the second series in order to avoid memory conflicts, and that processor i keeps track of l'_{i-1} and l'_{i+1}, relying on broadcasting by the leftmost processor if degeneracy in an interval occurs). When processor i has finished decrementing its two pointers in this fashion, a task requiring

$O(n/k)$ time and $O(k)$ extra space, the value of its second series pointer is its displacement table entry, E_i.

Thus displacement computing can be accomplished in $O(n/k + \log n)$ time and constant extra space per processor.

Series Splitting. At this point, processor i can easily determine from the entries in the displacement table the number of its records that are to be displaced to the block to its right (E_i), as well as the number of records that it is to receive from the block to its left (E_{i-1}) and from the second series ($E_i - E_{i-1}$). Thus we now seek to split, in parallel, the second series among the blocks of the first series. We accomplish this efficiently in constant extra space with the use of block rotations (each of which is effected with a sequence of three sublist reversals), followed by the desired data movement, followed by one last reversal. We illustrate this procedure in Figure 5, with the aid of some additional notation.

INSERT FIGURE 5

Letting i denote the index of an arbitrary processor with records in the first series only, we use X_i to denote its first $n/k - E_i$ records (that is, those to remain in this block) and Y_i to denote the remaining E_i records (that is, those to be displaced to the right). We use Z to denote the contents of the portion of a block that constitutes the second series. Processor i first reverses X_i and Y_i together, then each separately, thereby completing the rotation. Processor i then initiates data movement, employing a single extra storage cell to copy safely the last record of Y_i to the location formerly occupied by the last record of Y_{i+1}. We deviate from this if processor i is handling the last block of the first series, instructing it instead to copy its last Y record to the former location of the first Z record. At the same time, the processor of the second series copies its first Z record to the former location of the last Y record of the first (portion of a) block in the first series. Continuing in this fashion, therefore, the data movement sequence is right-to-left for the blocks in the first series, but left-to-right for the second.

Of course, when block i of the first series is filled, the processor of the second block must shift its attention to block $i+1$, and so on. If k is small enough (no greater than $O(\log n)$), then the displacement table can simply be searched; if k is larger than this, then the table may contain too many identical entries, and we invoke a preprocessing routine to condense it (again with the aid of broadcasting). The timing of the first and second series operations are interleaved (rather than simultaneous), because some processors will in general be handling portions of blocks of both types of series.

When the data movement phase is finished, each block will contain the correct prefix from the opposite series, but in reverse order. A final subblock reversal completes this step.

Series splitting, therefore, requires $O(n/k + \log n)$ time and constant extra space per processor.

Local Merging. We employ the aforementioned linear-time, in-place sequential merge from [HL1]. The completion of this merge is depicted in Figure 6.

INSERT FIGURE 6

Thus this final step requires $O(n/k)$ time and constant extra space per processor.

Implementation Details. Although the details necessary to handle lists and sublists of arbitrary sizes is the most intricate part of the sequential method, these details are quite simple for our parallel algorithm. We first fragment the input list $L = L1\ L2$ into the form $L3\ L4\ L5\ L6$, where both $L3$ and $L5$ contain an integral multiple of n/k records, and where $L4$ and $L6$ each contain strictly less than n/k (even, possibly, zero) records. With parallel rotations, it is easy to transform the list into the form $L3\ L5\ L4\ L6$ assuming the tail of $L4$ is less than or equal to the tail of $L6$ (or the form $L3\ L5\ L6\ L4$ if it is greater). We now invoke the main parallel algorithm on $L3\ L5$, yielding the sorted sublist $L7$. Ignoring obvious ways to streamline the remainder of this procedure, it is sufficient at this point merely next to invoke the sequential algorithm on $L4\ L6$, yielding the sorted sublist $L8$. Thus $L8$ can be viewed as at most one block of size n/k followed by at most one block of size strictly less than n/k. We now complete the merge by invoking the main parallel algorithm on $L7\ L8$, with every processor except possibly the last handling a block of size n/k. Even though the last block may have an unusual size at this step, it causes no problems for the main algorithm because its (large) tail ensures that it need not be moved during block sorting and because its (rightmost) position ensures that it need not be treated as a member of a first series when any pair of series is merged.

The time and space requirements necessary for implementation details are therefore bounded by those of the main parallel algorithm.

This completes the description of our parallel method. In summary, the total time spent is $O(n/k + \log n)$ and the total extra space used is $O(k)$. Therefore, this method is time-space optimal for any value of $k \leq n/(\log n)$, thereby meeting our stated goal.

4. Insuring Stability

It is often desirable that merging (and sorting) algorithms be *stable*, by which we mean that records with identical keys retain their original relative order after the algorithm is completed. Stability is a property that has extracted a heavy price in terms of increased complexity for sequential algorithms that operate in both optimal time and space simultaneously. The linear-time, in-place stable sequential merging algorithm with the lowest currently-known worst-case constant of proportionality is presented in [HL2], and is based largely on the unstable method that proved useful in guiding our thinking in devising the parallel algorithm presented in the last section. As one rough measure of the intricacy required to ensure stability in a sequential setting, we note that the worst-case constant of proportionality jumps from $3.125n$ (plus lower order terms) for the unstable algorithm of [HL1] to $7n$ (plus lower order terms) for the stable scheme of [HL2], where these values reflect an upper bound on the number of key comparisons plus record exchanges required.

Fortunately, however, the parallel procedure we have already presented can be made stable with relatively little effort. The only unstable routines in our main algorithm are found in the steps for block sorting, displacement computing and local merging. Instability in the block sorting step can be remedied by stabilizing the bitonic merge. To accomplish this, we need only specify that the block indices (which are already available) are to be used as "tie breakers" whenever equal tails are compared. The displacement computing step can be modified in a similar manner, by first stabilizing the bitonic merge (indices and offsets are already available) and then handling the two (now asymmetric) types of pairs of series in slightly different fashions in that $L1$ records must now receive priority over $L2$ records. The local merging step is stabilized by replacing the relatively simple but unstable in-place algorithm with the more complicated but stable in-place scheme. Only an extra pointer is needed to stabilize the implementation details (in the event that the $L3$ and $L4$ sublists each have a copy of the same key).

5. Extensions to Sorting and Open Problems

In this paper, we have presented for the first time parallel merging algorithms that are asymptotically time-space optimal. Moreover, our methods assume only the EREW PRAM model. Although n must be large enough so that the inequality $k \leq n/(\log n)$ is satisfied for optimality, we observe that our algorithms are efficient[5] for any value of n, suggesting that they may have practical merit even for relatively small inputs. Also, for the sake of complete generality, our algorithms modify neither the key nor any other part of a record.

These time-space optimal parallel merging algorithms naturally lead to time-space optimal parallel sorting algorithms, providing improvements over the best previously-published PRAM methods designed for a bounded number of processors. For example, the recent EREW merging and sorting schemes proposed in [AS] (where the issue of duplicate keys is

[5]A parallel method is efficient if its speedup is within a polylogarithmic factor of the optimum.

not even addressed) are time optimal only for values of $k \leq n/(\log^2 n)$. More importantly, such schemes are not space optimal for *any* fixed k.

We note that, from a practical standpoint, more streamlined sorting implementations may be possible. It is known from [HL2] that methods exist by which the obvious merge sort strategy can be replaced with more sophisticated sorting schemes that exploit merging in nontrivial ways. In that setting, for example, the worst-case constant of proportionality of the direct merge sort strategy is lowered from $7n \log n$ (plus lower order terms) to $2.5n \log n$ (plus lower order terms). Whether these more complicated techniques can be efficiently parallelized remains an open question.

Finally, from a more purely theoretical perspective, one might ask whether our methods can be extended to sub-logarithmic time merging. Because $\Omega(\log n)$ time is known to be a lower bound for merging on an EREW PRAM, our algorithms are the best possible (to within a constant factor) for this model. Asymptotically faster time-space optimal algorithms may exist, however, for more powerful models. For example, it is an open question whether time-space optimal merging can be accomplished in $O(n/k + \log \log n)$ time on a CREW PRAM.

Acknowledgments

We wish to express our appreciation to Karl Abrahamson and Bing-Chao Huang for stimulating technical discussions related to this general subject, and to the two anonymous referees for comments that have helped to improve the presentation of these results.

References

[Ak] S. G. Akl, Parallel Sorting Algorithms, Academic Press, Orlando, FL, 1985.

[AKS] M. Ajtai, J. Komlos and E. Szemeredi, "An $O(n \log n)$ Sorting Network," *Proc. 15th ACM Symp. on Theory of Computing* (1983), 1-9.

[AS] S. G. Akl and N. Santoro, "Optimal Parallel Merging and Sorting Without Memory Conflicts," *IEEE Trans. on Computers* 36 (1987), 1367-1369.

[Ba] K. E. Batcher, "Sorting Networks and their Application," *Proc. AFIPS 1968 SJCC* (1968), 307-314.

[BB] P. Banerjee and K. P. Belkhale, "Parallel Algorithms for Geometric Connected Component Labeling Problems on a Hypercube," Technical Report, Coordinated Science Laboratory, University of Illinois, Urbana, IL, 1988.

[BDHM] D. Bitton, D. J. Dewitt, D. K. Hsiao and J. Menon, "A Taxonomy of Parallel Sorting," *Computing Surveys* 16 (1984), 287-318.

[BH] A. Borodin and J. E. Hopcroft, "Routing, Merging and Sorting on Parallel Models of Computation," *J. of Computer and System Sciences* 30 (1985), 130-145.

[BS] G. Baudet and D. Stevenson, "Optimal Sorting Algorithms for Parallel Computers," *IEEE Trans. on Computers* 27 (1978), 84-87.

[Co] R. Cole, "Parallel Merge Sort," *SIAM J. on Computing* 17 (1988), 770-785.

[HL1] B-C Huang and M. A. Langston, "Practical In-Place Merging", *Communications of the ACM* 31 (1988), 348-352.

[HL2] B-C Huang and M. A. Langston, "Fast Stable Merging and Sorting in Constant Extra Space," *Proc. 1989 Intl. Conf. on Computing and Information* (1989), 71–80.

[HL3] B-C Huang and M. A. Langston, "Stable Duplicate-Key Extraction with Optimal Time and Space Bounds," *Acta Informatica* 26 (1989), 473–484.

[HL4] B-C Huang and M. A. Langston, "Stable Set and Multiset Operations in Optimal Time and Space," *Proc. 7th ACM Symp. on Principles of Database Systems* (1988), 288-293.

[Kn] D. E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison Wesley, Reading, MA, 1973.

[Kr] C. P. Kruskal, "Searching, Merging and Sorting in Parallel Computation," *IEEE Trans. on Computers* 32 (1983), 942-946.

[KR] R. M. Karp and V. Ramachandran,"A Survey of Parallel Algorithms for Shared-Memory Machines," Technical Report, Computer Science Division, University of California, Berkeley, CA, 1988.

[LDM] S. Lakshmivarahan, S. K. Dhall, and L. L. Miller, "Parallel Sorting Algorithms," *Advances in Computers* 23 (1984), 295-354.

[RB] D. Rose and F. Berman, "Mapping with External I/O: A Case Study," *Proc. 1987 Intl. Conf. on Parallel Processing* (1987), 859-862.

[RS] S. Ranka and S. Sahni, "Image Template Matching on SIMD Hypercube Multicomputers," *Proc. 1988 Intl. Conf. on Parallel Processing* (1988), Vol. 3, 84-91.

[SV] Y. Shiloach and U. Vishkin, "Finding the Maximum, Merging and Sorting in a Parallel Computation Model," *J. of Algorithms* 2 (1981), 88-102.

[Ul] J. D. Ullman, Computational Aspects of VLSI, Computer Science Press, Rockville, MD, 1984.

[Va] L. G. Valiant, "Parallelism in Comparison Problems," *SIAM J. on Computing* 4 (1975), 349-355.

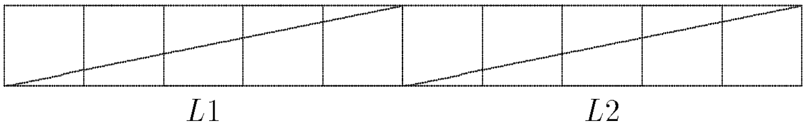

Figure 1. Divide Lists into Blocks

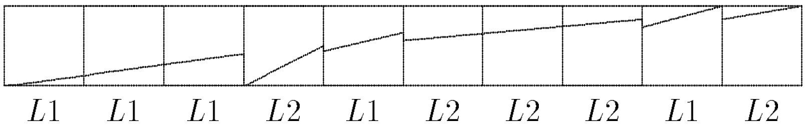

Figure 2. Block Sorting

$$\cdots\cdots\ \underbrace{\cdots\ 1\ 2}_{L2\ BLOCK}\ \underbrace{1\ \ \overbrace{2}^{breaker}\ \ 3\ 3\ 4\ 4}_{L1\ BLOCK}\ \underbrace{4\ 5\ 7\ 7\ 7\ 8}_{L1\ BLOCK}\ \underbrace{8\ 9\ 10\ 10\ 12\ 12}_{L1\ BLOCK}\ \underbrace{2\ 4\ 7\ 9\ \overbrace{13}^{breaker}\ \ 14}_{L2\ BLOCK}\ \cdots\cdots$$

a) Locating the Breakers

$$\cdots\cdots\ \ \cdots 1\ 2\ \ 1\ \underbrace{\overbrace{2}^{breaker}\ \ 3\ 3\ 4\ 4\ \ \ 4\ 5\ 7\ 7\ 7\ 8\ \ \ 8\ 9\ 10\ 10\ 12\ 12}_{first\ series}\ \underbrace{2\ 4\ 7\ 9}_{second\ series}\ \ \overbrace{13}^{breaker}\ \ 14\ \cdots\cdots$$

b) Pair of Resulting Series

Figure 3. Delimiting a Pair of Series to be Merged

$$\overbrace{\underbrace{2\ 3\ 3\ 4\ 4}_{f}\ \underbrace{4\ 5\ 7\ 7\ 7\ 8}_{f+1}\ \underbrace{8\ 9\ 10\ 10\ 12\ 12}_{f+2}}^{first\ series}\ \overbrace{\underbrace{2\ 4\ 7\ 9}_{f+3}}^{second\ series}$$

a) Pair of Series, p=3

i	E_i
f	1
f+1	2
f+2	4

b) Displacement Table Entries

Figure 4. A Pair of Series and the Corresponding Displacement Table Entries

$$\overbrace{\underbrace{\underbrace{2\ 3\ 3\ 4}_{X_1}\ \ \underbrace{4}_{Y_1}}_{f}\ \ \underbrace{\underbrace{4\ 5\ 7\ 7}_{X_2}\ \ \underbrace{7\ 8}_{Y_2}}_{f+1}\ \ \underbrace{\underbrace{8\ 9}_{X_3}\ \ \underbrace{10\ 10\ 12\ 12}_{Y_3}}_{f+2}}^{first\ series}\ \ \overbrace{\underbrace{\underbrace{2\ 4\ 7\ 9}_{Z}}_{f+3}}^{second\ series}$$

a) Notation

$$\overbrace{\underbrace{\underbrace{4}_{Y_1^R}\ \ \underbrace{4\ 3\ 3\ 2}_{X_1^R}}_{f}\ \ \underbrace{\underbrace{8\ 7}_{Y_2^R}\ \ \underbrace{7\ 7\ 5\ 4}_{X_2^R}}_{f+1}\ \ \underbrace{\underbrace{12\ 12\ 10\ 10}_{Y_3^R}\ \ \underbrace{9\ 8}_{X_3^R}}_{f+2}}^{first\ series}\ \ \overbrace{\underbrace{\underbrace{2\ 4\ 7\ 9}_{Z}}_{f+3}}^{second\ series}$$

b) Block Rotation

$$\overbrace{\underbrace{\underbrace{4}_{Y_1}\ \ \underbrace{2\ 3\ 3\ 4}_{X_1}}_{f}\ \ \underbrace{\underbrace{7\ 8}_{Y_2}\ \ \underbrace{4\ 5\ 7\ 7}_{X_2}}_{f+1}\ \ \underbrace{\underbrace{10\ 10\ 12\ 12}_{Y_3}\ \ \underbrace{8\ 9}_{X_3}}_{f+2}}^{first\ series}\ \ \overbrace{\underbrace{\underbrace{2\ 4\ 7\ 9}_{Z}}_{f+3}}^{second\ series}$$

c) Subblock Rotation

$$\underbrace{\underbrace{2}_{Z_1^R}\ \ \underbrace{2\ 3\ 3\ 4}_{X_1}}_{f}\ \ \underbrace{\underbrace{4}_{Z_2^R}\ \ \underbrace{4}_{Y_1}\ \ \underbrace{4\ 5\ 7\ 7}_{X_2}}_{f+1}\ \ \underbrace{\underbrace{9\ 7}_{Z_2^R}\ \ \underbrace{7\ 8}_{Y_2}\ \ \underbrace{8\ 9}_{X_3}}_{f+2}\ \ \underbrace{\underbrace{12\ 12\ 10\ 10}_{Y_3^R}}_{f+3}$$

d) Data Movement

$$\underbrace{\underbrace{2}_{Z_1}\ \ \underbrace{2\ 3\ 3\ 4}_{X_1}}_{f}\ \ \underbrace{\underbrace{4}_{Z_2}\ \ \underbrace{4}_{Y_1}\ \ \underbrace{4\ 5\ 7\ 7}_{X_2}}_{f+1}\ \ \underbrace{\underbrace{7\ 9}_{Z_2}\ \ \underbrace{7\ 8}_{Y_2}\ \ \underbrace{8\ 9}_{X_3}}_{f+2}\ \ \underbrace{\underbrace{10\ 10\ 12\ 12}_{Y_3}}_{f+3}$$

e) Subblock Rotation

Figure 5. Series Splitting

$$\underbrace{\underbrace{2}_{Z_1}\ \underbrace{2\ 3\ 3\ 4}_{X_1}}_{f}\ \underbrace{\underbrace{4}_{Z_2}\ \underbrace{4}_{Y_1}\ \underbrace{4\ 5\ 7\ 7}_{X_2}}_{f+1}\ \underbrace{\underbrace{7\ 9}_{Z_2}\ \underbrace{7\ 8}_{Y_2}\ \underbrace{8\ 9}_{X_3}}_{f+2}\ \underbrace{\underbrace{10\ 10\ 12\ 12}_{Y_3}}_{f+3}$$

a) Series Splitting Completed

$$\underbrace{2\ 2\ 3\ 3\ 4}_{f}\ \underbrace{4\ 4\ 4\ 5\ 7\ 7}_{f+1}\ \underbrace{7\ 7\ 8\ 8\ 9\ 9}_{f+2}\ \underbrace{10\ 10\ 12\ 12}_{f+3}$$

b) Local Merging

Figure 6. Local Merging