$n/(\log n)$ for optimality, we observe that our algorithms are also *efficient* in the usual sense (their speedup is within a polylogarithmic factor of optimal) for any value of n, suggesting that they may have practical merit even for relatively small files. As long as main memory remains a critical resource in many environments, the quest for techniques that permit the efficient use of both time and space continues to be a fertile research domain.

References

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

[GL] X. Guan and M. A. Langston, "Time-Space Optimal Parallel Merging and Sorting," *IEEE Transactions on Computers* 40 (1991), 596–602.

[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, "Stable Duplicate-Key Extraction with Optimal Time and Space Bounds," *Acta Informatica* 26 (1989), 473–484.

[HL3] B-C Huang and M. A. Langston, "Stable Set and Multiset Operations in Optimal Time and Space," *Proceedings, 7th ACM Symposium on Principles of Database Systems* (1988), 288–293.

Therefore, this select algorithm is time-space optimal for any value of $k \leq n/(\log n)$, thereby meeting our stated goal.

4. Time-Space Optimal Parallel Set Operations

In what follows, suppose we are given the input list $L = XY$, where X and Y are two sublists, each sorted on the key, and each containing no duplicates. Since the same key may naturally appear once in X and once in Y, we insist that, in the spirit of stability, the record represented in the result of a binary set operation be the one that occurs first in L.

We now have stable, time-space optimal parallel subroutines sufficient to perform the elementary binary set operations. Select is obtained from the work of the last two sections. Merge is obtained from [GL]. Duplicate-key extract is obtained from an easy modification to select, in which we replace the first step, local selecting, with the local duplicate key extracting method of [HL2]. (Local duplicate-key extract is actually easier than local select, because the $L1$ processors need no information from the $L2$ list.)

We invoke merge followed by duplicate-key extract to produce $X \cup Y$. We perform select to yield both $X \cap Y$ and $X - Y$. To achieve $X \oplus Y$, we invoke select on XY producing X_1X_2Y, rotate X_2 and Y to yield X_1YX_2, perform select on YX_2 producing $X_1Y_1Y_2X_2$, and finally merge X_1 and Y_1.

5. Concluding Remarks

Assuming only the weak EREW PRAM model, we have presented for the first time parallel algorithms for the elementary binary set operations that are asymptotically time-space optimal. As a bonus, these methods immediately extend to multisets (under several natural definitions [HL3]). Although n must be large enough to satisfy the inequality $k \leq$

block, and to set aside a one-bit otherwise. The processors now need only to compute prefix sums on these values, and then to acquire their respective new blocks in parallel without memory conflicts. This completes the block rearranging step, and has required $O(n/k+\log k)$ time and constant extra space per processor.

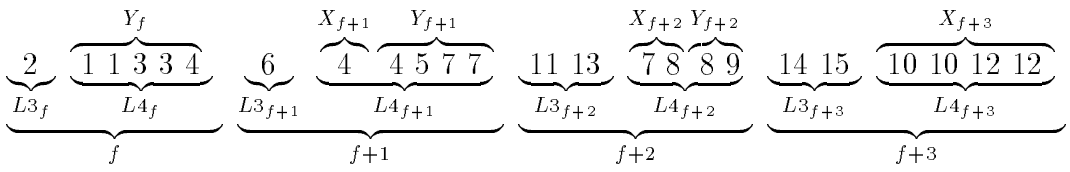
3. Implementation Details.

Suppose that the number of $L3$ (and hence $L4$) records is not evenly divisible by k, in which case the last breaker begins a series with strictly fewer than n/k $L3$ records. This series is treated just as any other (although it may be a very short one, lying entirely within the last block). In blockifying, the $L3$ records in this series are collected in the last block, so that this series becomes a (possibly empty) collection of $L4$ blocks followed by (possibly just part of) one block containing both $L3$ and $L4$ records. After the block rearranging step, we need only to move the $L3$ and $L4$ segments from the last block into their appropriate final positions with parallel rotations.

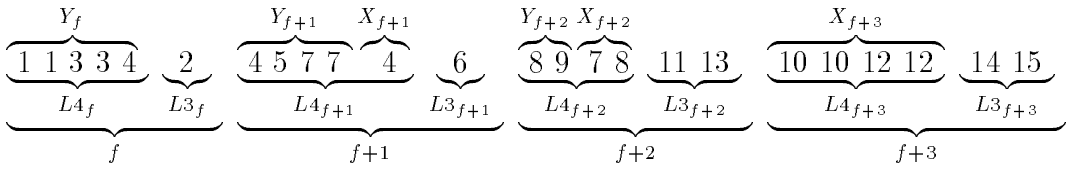
More generally, suppose that the number of $L1$ records is not evenly divisible by k. (Note that the number of $L2$ records never really needs to be evenly divisible by k.) We transform $L1L2$ into the list $L1^1L1^2L2$, where $L1^1$ contains an integral multiple of n/k records, and where $L1^2$ contains strictly fewer than n/k records. We further transform the input into the list $L1^1L2L1^2$ by means of parallel rotations, and invoke the main algorithm on $L1^1L2$, yielding $L3^1L4^1L2L1^2$. Then a local select of $L1^2$ against $L2$ gives $L3^1L4^1L2L3^2L4^2$. Parallel rotations now produce the desired result $L3^1L3^2L4^1L4^2L2 = L3L4L2$.

The time and space requirements of these implementation details are thus 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 $O(k)$.

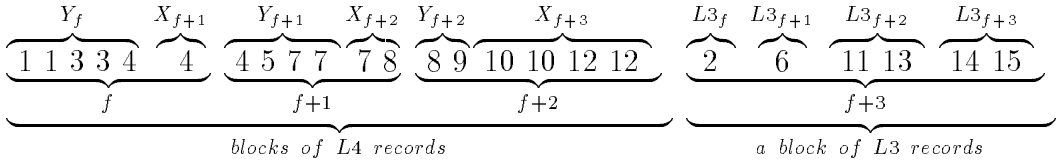
If $p = 2$, then the two blocks have the form $L3_i L4_i L3_{i+1} L4_{i+1}$, where $|L4_i| = |L3_{i+1}|$. Swapping $L4_i$ with $L3_{i+1}$ finishes the blockifying for this sequence. If $p > 2$, then we simply treat the sequence as we earlier did each series, exchanging the roles of $L3$ and $L4$ records. This completes the blockifying step, and has required $O(n/k + \log n)$ time and constant extra space per processor.



Sample Series



Subblock Permutations



Data Movement

Figure 3. Coalescing the $L3$ Records of One Series into a Single Block

Block Rearranging. $L3$ has now become an ordered collection of blocks interspersed with another ordered collection that constitutes $L4$. We need only to rearrange these blocks so that $L3$ is followed by $L4$. We direct each processor to set aside a zero-bit if it contains an $L3$

To accomplish the data movement, each processor first reverses the contents of its block, then reverses its X, Y and $L3$ segments separately, thereby efficiently permuting its (two or) three subblocks. Each processor j, $f < j \leq g$, now employs a single extra storage cell to copy safely the first record of X_j to the location formerly occupied by the first record of X_{j-1}, while processor f copies the first record of its $L3$ segment to the location formerly occupied by the first record of X_g. Data movement continues in this fashion, with each processor moving its $L3$ records to block g as soon as its X segment is exhausted.

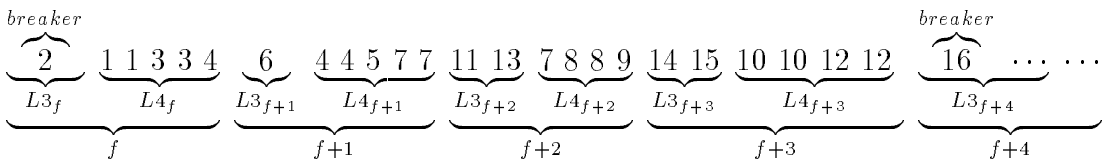
Note that if k is small enough (no greater than $O(max\{n/k, \log n\})$), then the displacement table can merely be searched; if k is larger than this, then the table may contain too many identical entries, and we invoke a preprocessing routine to condense it (again with the aid of broadcasting).

After the data movement is finished, it is necessary to rotate $L3_g$ with the records moved into block g from block $g+1$. The processing of the series is now completed, as depicted in Figure 3.

If block $g+1$ contains a leading breaker, we next rotate the records in an appropriate prefix of this block to ensure that $L3$ records precede $L4$ records there.

We can now handle the records not spanned by a series. These records are contained in zero or more non-overlapping "sequences" (we choose this term to avoid confusion with "series"), where each sequence begins with a leading breaker and ends with the record immediately preceding the next trailing breaker. Suppose such a sequence spans p blocks. Because there are exactly p breakers in these blocks, and because the $L3$ records before the first breaker and after the last breaker have been moved outside these blocks, there are now exactly $(p-1)(n/k)$ $L3$ records there. Thus, there are exactly n/k $L4$ records there.

algorithm proceeds. In Figure 2, our sample series is shown in more detail (with g set at $f+3$) along with its corresponding displacement table.



Sample Series

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

Displacement Table

Figure 2. A More Detailed View of a Sample Series and its Displacement Table

Thus each processor i, $f < i < g$, now uses the displacement table to determine exactly how the records in its block are to be rearranged: it is to send $|L3_i|$ records to block g, send its first E_{i-1} $L4$ records (which we denote by X_i) to block $i-1$, retain its next $n/k - |L3_i| - E_{i-1}$ $L4$ records (denoted by Y_i) and receive E_i $L4$ records (denoted by X_{i+1}) from block $i+1$. Processors f and g determine similar information: processor f is to send $|L3_f| = E_f$ records to block g and receive the same number of records from block $f+1$; processor g is to send $|L4_g| = E_{g-1}$ records to block $g-1$ and receive the same number of records from blocks f through $g-1$. (Note that segments X_f and Y_g are empty.)

follow it.

$$\cdots \underbrace{L3_f^- L4_f \; L3_{f+1} L4_{f+1} \cdots \; L3_{g-1} L4_{g-1} \; L3_g L4_g \; L3_{g+1}^-}_{one\ series} \cdots$$

Figure 1. A Sample Series Obtained in the Series Delimiting Step.

A processor that holds a lone or trailing breaker broadcasts its breaker's location to its right. After that, a processor that holds a lone or leading breaker broadcasts its breaker's location to its left. (This type of broadcasting can be efficiently accomplished on the EREW PRAM with data distribution algorithms or parallel prefix computations.) By this means, a processor learns the location of the lone or trailing breaker to its immediate left and the location of the lone or leading breaker to its immediate right. This completes the series delimiting step, and has required $O(\log(n/k) + \log k))$ time and constant extra space per processor.

Blockifying. In this step, we first reorganize in parallel the $L1$ records within every series, then reorganize in parallel the records in the remainder of the $L1$ list.

Let us consider our sample series as depicted in Figure 1. We seek to collect the n/k $L3$ records in this series in block g (and thus move the $L4$ records into the other blocks and subblocks illustrated). It is a simple matter to exchange $L3_{g+1}^-$ with the rightmost $|L3_{g+1}^-|$ records in $L4_g$. Efficiently coalescing the other $L3$ records into block g is much more difficult. We begin by computing prefix sums on $|L3_f^-|$, $|L3_{f+1}|$, $\cdots$, $|L3_{g-2}|$, $|L3_{g-1}|$ to obtain a "displacement table." Table entry $E_i = \sum_{k=f}^{i} |L3_k|$ denotes the number of $L3$ records in blocks indexed f through i that are to move to block g. It turns out that E_i will also denote the number of $L4$ records that block i is to receive from block $i+1$ as our

of the corresponding $L2$ processors to preprocess their records (performing the time-space optimal sequential select against the $L1$ block, followed by a time-space optimal sequential duplicate-key extract [HL2]), then instruct the $L1$ processor to perform its select (at most n/k $L2$ records are now needed), and finally direct the $L2$ processors to restore their blocks (two time-space optimal sequential merge operations suffice).

Thus, letting h denote the number of blocks in $L1$, the $L1$ list has now taken on the form $L3_1L4_1L3_2L4_2...L3_hL4_h$. This completes the local selecting step, and has required $O(n/k + \log n)$ time and constant extra space per processor.

Series Delimiting. We now seek to divide $L1$ into a collection of non-overlapping "series," each series with n/k $L3$ records. To begin this process, we locate special records that we term "breakers," each of which is the $(m(n/k)+1)$th $L3$ record for some integer m. First we compute prefix sums on $|L3_i|$ to find these breakers. For example, if $\sum_{i=1}^{g-1} |L3_i| < m(n/k)+1$ and $\sum_{i=1}^{g} |L3_i| \geq m(n/k)+1$, then block g contains the mth breaker. We identify three special types of breakers. If block i contains a breaker, but neither block $i-1$ nor block $i+1$ contain breakers, then the breaker in block i is called a "lone" breaker. If block $i-1$ and block i both contain breakers, and if block $i+1$ does not contain a breaker, then the breaker in block i is called a "trailing" breaker. If block i and block $i+1$ both contain breakers, and block $i-1$ does not contain a breaker, then the breaker in block i is called a "leading" breaker.

These breakers are used to divide $L1$ into non-overlapping series as follows: each series begins with a lone or trailing breaker and ends with the record immediately preceding the next lone or leading breaker. By design, each series contains exactly n/k $L3$ records. A sample series is depicted in Figure 1, where we use $L3_f^-$ to denote $L3_f$ minus any records that precede its breaker and $L3_{g+1}^-$ to denote $L3_{g+1}$ minus its breaker and any records that

2. Time-Space Optimal Parallel Select on the EREW PRAM Model

Given two sorted lists $L1$ and $L2$, our goal is to transform $L1$ into two sorted sublists $L3$ and $L4$, where $L3$ consists of the records whose keys are not found in $L2$, and $L4$ consists of the records whose keys are. Thus we accept $L = L1L2$, and *select* records from $L1$ whose keys are contained in $L2$, accumulating them in $L4$, where our output is of the form $L3L4L2$.

Our parallel algorithm comprises four steps: *local selecting, series delimiting, blockifying* and *block rearranging*. To facilitate discussion, we temporarily assume that the number of records of each type ($L1$, $L2$, $L3$ and $L4$) is evenly divisible by k, where k denotes the number of processors available.

Local Selecting. We first view L as a collection of k blocks, each of size n/k, and associate a distinct processor with each block. We seek to treat each $L1$ block $L1_i$ as if it were the only block in $L1$, transforming its contents into the form $L3_iL4_i$.

Our first task in this step is to determine where each tail (rightmost element) of each $L1$ block would go if the tails alone were to be merged with $L2$. In order to make this determination efficiently on the EREW model, we direct each $L1$ processor to set aside four extra storage cells (for copies of indices, offsets and keys) and employing the "phased merge" as described in the displacement computing step of the merge in [GL]. At most $O(\log n)$ time and $O(k)$ extra space has been consumed up to this point.

As long as an $L1$ processor doesn't need to consider more than $O(n/k)$ $L2$ records (a quantity known by considering the difference between where its block's tail would go and where the tail of the block to its immediate left would go if they were to be merged with $L2$), we instruct it to employ the linear-time, in-place sequential select routine from [HL3]. Otherwise, in the case that an $L1$ block spans several $L2$ blocks, we first enlist the aid

1. Introduction

The design and analysis of optimal parallel file rearrangement algorithms has long been a topic of wide-spread attention. The vast majority of the published literature has concentrated on the search for algorithms that are *time optimal*, that is, those that achieve optimal speedup (see, for example, [AS]). Unfortunately, space management issues have often taken a back seat in these efforts, leaving those who seek to implement optimal parallel algorithms unable to do so with any reasonable, bounded number of processors.

In recent work, however, parallel merge and sort methods that simultaneously optimize *both* time and space have been devised [GL]. Such *time-space optimal* algorithms attain optimal speedup, yet require only a constant amount of extra space per processor, even when the number of processors is fixed. Just what scope of file rearrangement problems is amenable to time-space optimal parallel techniques? In this paper we provide a partial answer to this question, developing time-space optimal parallel algorithms for the elementary binary set operations, namely, set union, intersection, difference and exclusive or.

To accomplish our goal, we devise a new parallel *select* procedure, reducing the general problem to one of a series of disjoint local operations, one for each processor, on which we can exploit sequential methods. Given an EREW PRAM with k processors, our algorithms operate on two sorted lists of total length n in $O(n/k + \log n)$ time and $O(k)$ extra space, and are thus time-space optimal for any value of $k \leq n/(\log n)$. For the sake of complete generality, our algorithms are stable (records with identical keys retain their original relative order), do not modify records (even temporarily) as they execute, and require no information other than a record's key.

Symbols Used

O capital Greek omicron of "big oh" notation

Σ capital Greek sigma for summations

$\cup$ set union

$\cap$ set intersection

$\oplus$ set exclusive or

Running Head

Parallel File Rearrangement Methods

Communicating Author

Michael A. Langston, (615) 974–3534, langston@cs.utk.edu

Abstract

We present parallel algorithms for the elementary binary set operations that, given an EREW PRAM with k processors, operate on two sorted lists of total length n in $O(n/k + \log n)$ time and $O(k)$ extra space, and are thus time-space optimal for any value of $k \leq n/(\log n)$. Our methods are stable, require no information other than a record's key, and do not modify records as they execute.

Accepted for Publication in *Journal of Parallel and Distributed Computing*

Parallel Methods for Solving Fundamental File Rearrangement Problems*†

Xiaojun Guan
Engineering Physics and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN 37831

Michael A. Langston
Department of Computer Science
University of Tennessee
Knoxville, TN 37996

---

* A preliminary version of a portion of this paper was presented at the International Conference on Databases, Parallel Architectures and Applications (PARBASE–90), held in Miami Beach, Florida, in March 1990.

† This research has been supported in part by the National Science Foundation under grant MIP–8919312 and by the Office of Naval Research under contract N00014–90–J–1855.