Parallel Search in Matrices with Sorted Columns*

Amit Jain
Department of Mathematics and Computer Science
Boise State University
Boise, Idaho 83725
email: amit@cs.idbsu.edu

Abstract

Abstract

In this paper we consider searching, and also ranking, in an $m \times n$ matrix with sorted columns on the EREW PRAM model. We propose a work-optimal parallel algorithm, based on the technique of accelerated cascading, that runs in $O(\log m \log\log m)$-time for small elements with rank $k \leq m$ and in $O(\log m \ \log\log m \ \log(k/m))$-time otherwise. Then we present a sequential algorithm for multisearch in a matrix with sorted columns as a prelude to a parallel algorithm for multisearch in a matrix with sorted columns. The sequential algorithm uses ideas from the parallel technique of chaining. The parallel multisearch algorithm follows this sequential algorithm and has a nontrivial dependence not only on the ranks of the search-elements but also on the number of search-elements. Finally we show how to adapt ideas from Bentley and Yao's [2] paper on sequential unbounded searching to parallel searching in matrices, which surprisingly leads to an asymptotic improvement.

1 Introduction

In this paper we are interested in parallel search and multisearch algorithms for matrices with sorted columns on the Exclusive Read Exclusive Write Parallel Random Access Machine model (EREW PRAM) [14]. Given an $m \times n$ matrix with sorted columns or, equivalently, m sorted arrays of size n each, we consider the problem of searching for a given value z and computing its rank k among the elements of the matrix M. Then we consider a generalization— multisearch. The problem of multisearch consists of ranking r input elements $z_1 < z_2 < \ldots < z_r$ in the $m \times n$ matrix M with sorted columns. Search and selection in such matrices have received attention because of their applications in statistics, operations research and combinatorics, among others [9, 10, 13]. Sequential search algorithms have been designed to exploit the constraints placed on these sets [9].

We mention two additional applications of the problem that we are considering. Firstly, our searching algorithms may be used for searching through a collection of extremely large ordered tables, such as in databases, if one wished to pay a search cost proportional to the item's distance from the front of the tables. The second application occurs in testing a system, consisting of m parallel subsystems, for a breaking point, which might involve an unbounded search over m sets of values.

We employ the technique of accelerated cascading [7, 12] along with that of chaining [16, 12] to design parallel algorithms for searching in matrices with sorted columns. The technique of chaining alone does not lead to fast and efficient algorithms since these matrices are partial orders. In addition, our results lead to an interesting parallel analog of Bentley and Yao's result on sequential unbounded searching [2].

The reader is referred to the book by Ja′Ja′ [12] for background and terminology on parallel algorithms. We say that a parallel algorithm is *work-optimal* if the total number of operations performed has the same order as that for an optimal sequential algorithm (or the best known sequential algorithm if the sequential complexity is not known). We say a parallel algorithm is *optimal* if it is work-optimal and has the fastest possible running time.

The paper is organized as follows. Section 2 describes the parallel algorithm for search in a matrix with sorted columns. In Section 3, the results of Section 2 are generalized to the multisearch problem. In Section 4, an almost optimal parallel algorithm is pre-

*Part of this work was done while the author was at the University of Central Florida, supported by NSF Grant CDA-9115281

sented for search in a matrix with sorted columns. Finally, we conclude in Section 5 by mentioning some open problems arising from this work.

2 Search in Matrices with Sorted Columns

First, let us examine multiple binary search in a single sorted array of size n since later we will use this search as a subroutine. Using the technique of chaining, Chen has shown how to perform multiple binary searches efficiently. His result is stated in the following theorem.

Theorem 2.1 (Chen [5]) *Given a sorted array of size n, elements $z_1 < z_2 < \ldots < z_r$ can be searched for in $O(\log n + \log r)$ time using r processors on the EREW PRAM.*

Chen also gave another algorithm for multiple search that is more complicated but optimal. We will use the sequential simulation of this parallel algorithm later in the paper.

Theorem 2.2 (Chen [5]) *Given a sorted array of size n, elements $z_1 < z_2 < \ldots < z_r$ can be searched for in $O(\log n + \log r)$ time using $r \log(n/r)/\log n$ processors on the EREW PRAM.*

The problem of searching for z in a matrix with sorted columns was solved sequentially by Frederickson and Johnson [9]. Here we sketch their optimal sequential algorithm. The algorithm consists of performing an one-sided binary search, proposed by Bentley and Yao [2], on each column to find the insertion point in each column. If the value z is found during this procedure then the search is done, otherwise the one-sided binary search identifies an interval in which z may lie. This phase of the one-side binary search is just a linear search over the elements $1, 2, 4, 8, \ldots, 2^i$. Next, ordinary binary search is used on this interval either to find the value or to locate the insertion point. Let the insertion point in the jth column be k_j, where $k_j > 0$. In $O(m)$ time, we can easily determine k_js that are non-zero and then carry out the search only in those columns. Of course, if the rank $k < m$ then at most k columns will be selected. The time required for search in each column is $O(\log k_j)$ and, if $p = \min\{k, m\}$, the total time required by the algorithm is

$$\sum_{j=1}^{p} \log k_j = \log(k_1 k_2 \cdots k_p) \leq \log \left(\frac{k}{p}\right)^p = O(p \log \frac{k}{p}).$$

Frederickson and Johnson [9] also proved a corresponding lower bound. To the best of our knowledge no one has reported a parallel algorithm for search or multi-search on matrices with sorted columns.

For ease of exposition we consider two cases:(i) the rank $k \leq m$, and (ii) the rank $k > m$. In the first case the time taken by the optimal sequential algorithm is $O(m)$, and in the second case the time taken is $O(m \log(k/m))$. In the extreme case when $k = \Theta(mn)$, the time taken is $O(m \log n)$. Note that the technique used by Sarnath and He [17] for designing a parallel algorithm for searching in matrices with sorted rows and sorted columns is not applicable here since the rows are not sorted. A straightforward approach like using m processors and performing a binary search in each row in $O(\log n)$ time is fast but not work-optimal except in the extreme case of the rank being $\Theta(mn)$. If the rank of the element z is less than n, then we could improve the cost of this approach slightly to $O(m \log k)$ by using the one-sided binary search in each row. But that is still not work-optimal as many processors will be idle in rows where the one-sided search ends much sooner than in other rows. We will attempt to develop a work-optimal parallel algorithm by reducing the number of processors to $m/\log m$.

Consider the first case, when $k \leq m$. Since we have $m/\log m$ processors, the natural approach seems to be to divide the columns into blocks of $\log m$ columns so that one processor is assigned to perform the one-sided binary search in each block of columns. Since there is no relative ordering among the rows, all of the search may be concentrated in one or a few blocks leading to little or no improvement. Instead we will use the idea of *accelerated cascading* of two algorithms, one work-optimal but relatively slow, the other fast but not work-optimal. The main idea is to proceed with the one-sided binary searches in parallel in each column until the number of unterminated searches is $O(m/\log m)$ and then switch to the fast algorithm where one processor is assigned to each of the remaining columns.

Initially, the first element in each column is examined in $O(\log m)$ time using $m/\log m$ processors. This corresponds to the first step of the one-sided binary search in each column. Now let r_1 be the number of unterminated searches. The algorithm proceeds in two phases.

Phase I. In each iteration use $m/\log m$ processors to simulate one step of the linear search part of the one-sided binary search; that is, examine elements at positions $2, 4, 8, \ldots, 2^i$ in each column. This requires $r_i/(m/\log m)$ steps where r_i is the number of unterminated searches before iteration i. Continue this phase until $r_i \leq m/\log m$. At this point the number of unterminated searches is less than or equal to the number of processors. Thus each processor can finish the one-sided binary search in $O(\log m)$ time.

Phase II. Reallocate the processors to the remaining intervals identified in Phase I so that each processor finishes the search in a section of $O(\log m)$ elements. Note that each section may span several columns.

At the end of Phase I we either have found the rank k_j of the element z in the jth column or have identified an interval to which the element z belongs. The size of the interval is less than $2k_j$. The total size of these intervals is $O(k)$ but there could be $O(k)$ such intervals. However, the size of any one interval could be as large as $2k$.

Since the rank $k \leq m$, the number of unterminated searches after initialization, r_1, is less than m. In iteration i of Phase I the 2^ith element is examined in each of the r_i columns where the search has not been terminated. Thus the value of r_i is less than $m/2^i$. Otherwise we are examining elements with rank greater than m, which is greater than k, a contradiction. Phase I ends when $r_i < m/2^i \leq m/\log m$, which implies that the number of iterations is $O(\log\log m)$. Furthermore the total work done during Phase I is given by

$$\sum_{i=1}^{\log\log m} \frac{m}{2^i} = O(m),$$

in the case when $k = m$ and by $O(k)$ if $k < m$. In each iteration, however, the $m/\log m$ processors have to be reassigned to those columns where the one-sided binary search has not finished. This processor scheduling and assignment requires a parallel prefix operation to compact the list of columns where the search has not yet finished. The parallel prefix can be computed in $O(\log m)$ time using $O(m)$ work [15]. Thus the overall time for Phase I is $O(\log m \log\log m)$.

For implementing Phase II we construct two arrays $WT[1..m]$, where $WT[j]$ is the number of elements left in the interval identified in the jth column, and $POS[1..m]$, where $POS[j]$ is a pair identifying the start and end of the interval in the jth column that still has to be searched. Note that the sum of the values in the array WT is $O(m)$, if $k \leq m$, and is $O(k)$ otherwise. In Phase II we want to divide these $O(m)$ elements equally among the $m/\log m$ processors for finishing the search. The Phase II can be performed as follows.

Phase II.a Compute the parallel prefix of the array $WT[1..m]$.

Phase II.b Perform multisearch for $i \times \log m$, where $1 \leq i \leq m/\log m$, in the array consisting of the partial sums for the array $WT[1..m]$. The multisearch results in column indices.

Phase II.c Let the ith processor, where $1 \leq i \leq m/\log m$, use simple linear search in its section of size $O(\log m)$ and, if desired, compute the rank.

All of these steps of Phase II can be finished in $O(\log m)$ time using $O(m)$ work. Overall the algorithm requires $O(\log m \log\log m)$ time, but since we are using $m/\log m$ processors the cost is not optimal. Since the total work done is only $O(m)$, we can use Brent's Scheduling [4] and reduce the number of processors to $(m/\log m \log\log m)$. Now the time required for Phase I will be

$$O(\log m \sum_{i=1}^{\log\log m} \frac{\log\log m}{2^i}) = O(\log m \log\log m).$$

Phase II will also require $O(\log m \log\log m)$ time using $m/\log m \log\log m$ processors. Thus we have the following result.

Theorem 2.3 *Given an $m \times n$ matrix with sorted columns, a value z, with rank $k \leq m$, can be searched for and its rank k computed in $O(\log m \log\log m)$ time and $O(m)$ work on the EREW PRAM.*

Now we will consider the case when the rank $k > m$. Note that we will detect this at the end of the Phase I by summing the weight array WT. Consider iteration i in Phase I. If the one-side binary search is active in all columns then we have examined $2^i m$ elements already. This can continue as long as $2^i m < k$, that is, the number of iterations is $O(\log(k/m))$. In each iteration we need $O(\log m)$ time to compact the list of unfinished searches using parallel prefix. Thus the overall time for this part of Phase I is $O(\log m \log(k/m))$ using $m/\log m$ processors. For the rest of Phase I of the algorithm we need to distinguish between two cases: (i) when $k/m \geq \log m$, and (ii) when $k/m < \log m$. If $k/m \geq \log m$, then in the next iteration there are at most $O(m/\log m)$ unfinished one-sided searches. These can be done in $O(\log n)$ time, in the worst-case, using $O(m \log n/\log m)$ work, which in this case is still $O(m \log(k/m))$. In addition, we will also have computed the rank of the element z in these columns. If $k/m < \log m$, then we may need another $O(\log\log m)$ iterations to finish the Phase I. So Phase I requires $O(\log m \log\log m + \log m \log(k/m)))$ time in this case and the work is $O(m \log(k/m) + m)$.

In Phase II we may have one interval each left in some of the columns after the one-sided binary search. The total number of elements in these intervals is given by $\sum_{j=1}^{m} WT[j]$, which is less than $2k$. If $k/m \geq \log m$, then the size of any interval is $O(k/m)$. Assigning one processor each to $\log m$ columns allows us to finish the search and the computation of the rank in $O(\log m \log(k/m))$ time. If however, $k/m < \log m$, then the size of the interval in each column could be as much as $O(\log m)$ and the above approach leads to a time of $O(\log m \log\log m)$ with $O(m \log\log m)$ cost, which is not optimal. Cost-optimality can be achieved by the following strategy. The sum of the entries in the WT array provides us with an estimate on the value of the rank k. Let this be denoted by k'. Take the first k'/m elements from each interval and use one processor to search $\log m$ columns in $O(\log m \log(k/m))$ time and optimal cost. Now the number of remaining elements are $O(m)$, so we can use the same approach as used in the case where rank $k \leq m$. This can be done in $O(\log m)$ time with $O(m)$ work.

Overall the algorithm runs in $O(\log m \log(k/m) + \log m \log\log m)$ time and requires $O(m \log(k/m))$ work. Applying Brent's scheduling, we can use $m/\log m \log\log m$ processors and the resulting algorithm is work-optimal for all values of the rank k.

Theorem 2.4 *Given an $m \times n$ matrix with sorted columns, a value z can be searched for and its rank k computed in $O(\log m \log\log m + \log m \log(k/m))$ time and $O(m + m\log(k/m))$ work on the EREW PRAM.*

3 Multisearch in Matrices with Sorted Columns

The problem of multisearch consists of ranking the elements $z_1 < z_2 < \ldots < z_r$ in the $m \times n$ matrix M with sorted columns. To avoid confusion we will refer to the elements $z_1, \ldots, z_r$ as the search-elements and the elements of the matrix M as merely elements. The number of search-elements, r, is assumed to be between 1 and mn. First we will consider sequential algorithms for the problem of multisearch that are sensitive to the number of search-elements and to the rank of the search-elements. Then we will present the parallel algorithm for multisearch in matrices with sorted columns, which is a generalization of our parallel algorithm for the search presented in the previous section.

3.1 Sequential Algorithm

Initially we can sort the first row of the matrix M in $O(m \log m)$ time and then merge the elements $z_1, \ldots, z_r$, in $O(m+r)$ time, with the sorted row so that we can determine which columns are to be searched. Now we can search for each element separately. The total time required by this simple approach is

$$O(m \log m + r + \sum_{i=1}^{r} m \log(K^{(i)}/m)),$$

where $K^{(i)}$ is the rank of the ith element z_i in the matrix M.

We can improve the above sequential algorithm as follows. First search for the largest element z_r so that we have a bound on how far to search for the rest of the elements in each column. This requires $O(m \log(K^{(r)}/m))$ time, where $K^{(r)}$ is the rank of the element z_r. Let the rank of the element z_r in column j be $K_j^{(r)}$. Note that since $z_1 < z_2 < \ldots < z_r$, we only search elements at indices no more than $K_j^{(r)}$ in column j.

In addition we also determine in what columns we should search for each of the elements, which is first step of the one-sided search. If the number of elements r is very small, then we could just check one at a time in $O(rm)$ time. Otherwise, we can first sort the matrix elements in the first row and then use the sequential multiple binary search algorithm [5] to search for the r elements in the m sorted elements of the first row. If $r \leq m$, then the total time for this initial step is $O(m \log m + r \log(m/r))$, which is $O(m \log m)$. Otherwise, if $r > m$, we can search for the m sorted elements in the r given elements in $O(m \log m + m \log(r/m))$-time.

After the one-sided binary search for z_r ends we have identified a set of elements of size at most $O(m)$ when the rank of z_r is less than or equal to m. In this case we can merge the remaining $r-1$ search-elements with at most $O(m)$ elements identified during the search for z_r, and we will have the ranks for all the search-elements.

In all other cases we use the optimal sequential algorithm for multiple binary search presented by Chen [5] on each column where the elements may be found. This algorithm requires $O(r \log(K_j^{(r)}/r))$ time for searching r elements in a sorted column of size $K_j^{(r)}$. Thus the total time for the algorithm is now

$$O(\sum_{j=1}^{m} r \log(K_j^{(r)}/r)),$$

which can be simplified to $O(rm \log(K^{(r)}/mr))$.

In case the number of elements r is greater than n or $K^{(r)} < rm$, we can change the search around such that we are searching for the elements of the column in the r given elements. The time-complexity then becomes

$$O(\sum_{j=1}^{m} K_j^{(r)} \log(r/K_j^{(r)})),$$

which can be simplified to

$$O(K^{(r)} \log(rm/K^{(r)})).$$

Note that the value of $K^{(r)} > m$ for this case and the value of $r > n$. These results are summarized in the next theorem.

Theorem 3.1 *Given r search-elements, $z_1 < z_2 < \ldots < z_r$, to search for in an $m \times n$ matrix M with sorted columns, the time-complexity of the sequential algorithm is:*

1. $1 < r \leq \log m$:
   (a) $K^{(r)} = O(m)$: $O(rm)$.
   (b) $K^{(r)} = \Omega(m)$ *and* $K^{(r)} < rm$: $O(rm + K^{(r)} \log(rm/K^{(r)}))$.
   (c) $K^{(r)} = \Omega(m)$ *and* $K^{(r)} > rm$: $O(rm + rm \log(K^{(r)}/rm))$.
2. $\log m < r \leq m$:
   (a) $K^{(r)} = O(m)$: $O(m \log m)$.
   (b) $K^{(r)} = \Omega(m)$ *and* $K^{(r)} < rm$: $O(m \log m + m \log(K^{(r)}/m) + K^{(r)} \log(rm/K^{(r)}))$.
   (c) $K^{(r)} = \Omega(m)$ *and* $K^{(r)} > rm$: $O(m \log m + m \log(K^{(r)}/m) + rm \log(K^{(r)}/rm))$.
3. $m < r \leq n$:
   (a) $K^{(r)} < rm$: $O(m \log r + m \log(K^{(r)}/m) + K^{(r)} \log(rm/K^{(r)}))$.
   (b) $K^{(r)} > rm$: $O(m \log r + m \log(K^{(r)}/m) + rm \log(K^{(r)}/rm))$.
4. $n < r \leq mn$: $O(m \log r + K^{(r)} \log(rm/K^{(r)}))$.

3.2 Parallel Algorithm

The parallel algorithm for multisearch is also based upon the number of elements we want to search for. Initially we run the parallel algorithm for single search and find the largest search-element z_r and its rank $K^{(r)}$, as well as its rank in each column, $K_j^{(r)}$, $1 \leq j \leq m$. The time taken depends upon the rank of z_r. Let us first consider the case $K^{(r)} \leq m$, which also implies that $r \leq m$ since all the search-elements are distinct. Similar to the sequential algorithm proposed in the previous section, the two cases have to be examined separately.

If $1 \leq r \leq \log m$, then after execution of the Phase II.a and II.b of the parallel algorithm for a single search we have identified a total of $O(m)$ elements within sections $[1 \ldots K_j^{(r)}]$, $1 \leq j \leq m$. Phase II.b for a single search also identifies sections of size $O(\log m)$, but now we may have multiple processors searching within these sections. Since $r \leq \log m$, we can simply pipeline the r searches through the section in time $O(r + \log m)$, or equivalently $O(\log m)$, making the total work $O(rm)$. Thus we have an $O(\log m \log\log m)$ time algorithm that performs $O(rm)$ work. Similar to the single search, we can apply Brent's Scheduling and reduce the number of processors to $rm/\log m \log\log m$ and keep the same running time.

If $\log m < r \leq m$ then after searching for z_r, we can simply merge the $O(m)$ elements with the r search-elements in $O(\log m)$ time using $m/\log m$ processors and thus compute all the ranks. We can use any optimal parallel merging algorithm on the EREW PRAM for this purpose (e.g. [3, 11, 8]). The total time is then $O(\log m \log\log m)$ and the total work is $O(m \log m)$.

Now we will consider the case when the rank $K^{(r)} > m \log m$, similar to the case for the parallel algorithm for a single search. At the end of the one-sided binary search for the search-element z_r we have narrowed the search to $m/\log m$ columns in $O(\log m \log(K^{(r)}/m))$ time and $O(m \log(K^{(r)}/m))$ work. At this point we will finish the search in these $m/\log m$ columns and know the rank of the search-element z_r in these columns. Thus we can perform multisearch for the rest of $r-1$ search-elements in $O(\log r + \max_j\{\log K_j^{(r)}\})$-time and the work performed is no more than that of the sequential algorithm. After this phase we must return to complete the search in the intervals identified during the initial one-sided search. In this case we will assign r processors to $\log m$ columns to the intervals identified by the search for the element z_r. From earlier arguments we know that the size of these intervals is $O(K^{(r)}/m)$. We will use the multisearch algorithm by Chen [5]. The multisearch in one column requires $O(\log(K^{(r)}/m) + \log r)$ time leading to overall time of $O(\log m(\log(K^{(r)}/m) + \log r))$. If $K^{(r)}/m < r$ then we can instead search for the at most $K^{(r)}/m$ sorted elements of a column in the r search-elements and the algorithm has the same cost as the sequential algorithm discussed in the previous subsection.

The other case to be considered is when $n < r \leq mn$. Then we use the multisearch algorithm to search for the elements of the column in the r search-elements. The time is same as the previous case and the work done is the same as in the sequential algorithm. Finally, when the rank $K^{(r)}$ is less than $m \log m$, which occurs only in the case $1 \leq r \leq \log m$, then we can split the elements as done in the case for the parallel algorithm for single search and then apply the two methods proposed above on parts of the problem.

4 Faster Parallel Search in Matrices with Sorted Columns

The parallel algorithm for searching in a matrix with sorted columns presented earlier can be made to run asymptotically faster while maintaining work-optimality. The algorithm in this section is inspired by the ideas contained in the paper by Bentley and Yao [2] on unbounded searching. Bentley and Yao present a series of algorithms for searching in an unbounded sorted array for an element with rank n. Their ultimate algorithm requires the following number of comparisons:

$$\log n + \log^{(2)} n + \log^{(3)} n + \ldots + 2\log^{(q)} n + 1,$$

where $\log^{(q)}$ is the log function applied q times to n. The value q is chosen such that $q = \min_q\{\log^{(q)} n \leq 1\}$, that is, $q = O(\log^* n)$. If we use the simple unbounded search then the number of comparisons is $(2\log n + 1)$, which differs from the ultimate algorithm only by a constant factor. Surprisingly, when we apply these ideas to searching in matrices with sorted columns the time turns out to be asymptotically faster.

Recall that we examine elements at index at most $\log m$ before the number of searches is reduced below $m/\log m$. Thus we can take n to be $\log m$. Instead of examining indices $1, 2, \ldots, 2^i$ we will instead examine indices as follows: $2, 2^2, 2^{2^2}, \ldots, 2^{2^{2^{\cdot^{\cdot^{2}}}}}$. Thus the number of iterations is $\log^*(\log m)$. After the one-sided search ends we have identified an interval. Within this interval we will now use ordinary binary search but over sparsely situated elements. The first time we examine only $\log^{(q)} m$ elements and then $\log^{(q-1)} m$ elements and so on until the last stage when we examine at most $\log m$ elements per column. Thus there are q stages where $q = O(\log^*(\log m))$.

Suppose that the rank of the element being searched for is $k \leq m$. Then in each iteration the number of unfinished searches is much less than half of that in the previous iteration. Therefore the $O(\log^*(\log m))$ iterations of the first stage can be implemented in $O(\log m \log^*(\log m))$ time and $O(m)$ work. In order to finish the search we must successively refine the interval until we are left with $O(m)$ elements. This is accomplished by the nested binary searches mentioned in the previous paragraph. The number of such searches is $O(\log^*(\log m))$. We will show that the number of elements examined in the ith nested binary search is no more than $m/2^i$, thus showing that the overall work is $O(m)$ and the time required is $O(\log m \log^*(\log m))$.

Let the rank of the element in column j be k_j. First we observe that

$$\sum_{j=1}^{m} \log k_j = O(m \log(k/m)),$$

which is $O(m)$ (since $k \leq m$). This is the case in the last stage. In the penultimate stage the total number of elements examined by the nested search is

$$\sum_{j=1}^{m} \log\log k_j \leq \sum_{j=1}^{m} (\log k_j)/2 = O((m \log(k/m)/2)),$$

or $O(m/2)$. Similarly, in the qth nested search before the last stage the total number of elements examined is $O(m/2^q)$. Thus we can search for a element with rank $k \leq m$ in $O(\log m \log^*(\log m))$ time and $O(m)$ work.

If the rank $k > m \log m$, then the number of iterations in the first stage is $O(\log^*(k/m))$ and the overall time is $O(\log m \log^*(\log m) \log(k/m))$ when we are using $m/\log m \log^*(\log m)$ processors. If the rank is intermediate, then we can use the strategy used before in Section 2 to obtain the time $O(\log m \log^*(\log m) \log(k/m) + \log m \log^*(\log m))$ with $O(m + m\log(k/m))$ work.

5 Further Directions

The following open problems arise from this work. Is it possible to obtain an $O(\log m)$-time and $O(m + m\log(k/m))$-work algorithm for searching in a matrix with sorted columns? Such an algorithm would be time-optimal and work-optimal. We were able to get within a factor of $O(\log^*(\log m))$ in this paper. Obtaining an optimal parallel algorithm for search in sorted matrices with sorted columns seems to be as hard as the optimal list ranking parallel algorithm for the EREW PRAM [7, 6, 1].

We have used the idea of multisearch in a sorted array as a subroutine to derive our results. The multisearch technique potentially has many other applications that can be investigated.

We can also investigate parallel selection in matrices with sorted columns. Frederickson and Johnson [9] have presented an optimal sequential algorithm for selection in a matrix with sorted columns that requires

$O(m+p\log(k/p))$-time, with $p=\min\{k,m\}$. Their algorithm can be parallelized relatively easily to run in $O(\log m \log(k/m) \log\log m)$ time with optimal work. However it is not clear if this time can be improved. For matrices with sorted columns and sorted rows, Sarnath and He [17] have presented a parallel algorithm for selection in an $n \times n$ matrix with sorted rows and sorted columns that runs in $O(\log n \log\log n \log^* n)$ time with $O(n \log\log n)$ work on the EREW PRAM. Their techniques do not apply to matrices with only columns sorted as these matrices have less structure.

Acknowledgements: I would like to thank Professor Narsingh Deo for his comments and encouragement. I would like to thank Professor Robert Sulanke and Mike Stark for their help in improving the presentation of the paper.

References

[1] Richard J. Anderson and Gary L. Miller. Deterministic parallel list ranking. In *VLSI Algorithms and Architectures, LNCS, 319*, pages 81–90. Springer-Verlag, 1988.

[2] J. L. Bentley and A. C. Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5:82–87, 1976.

[3] Gianfranco Bilardi and Alexandru Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared memory machines. *SIAM Journal on Computing*, 18(2):216–228, April 1989.

[4] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974.

[5] Danny Z. Chen. Efficient parallel binary search on sorted arrays. Technical Report 1009, Purdue University, Department of Computer Science, August 1990.

[6] Richard Cole and Uzi Vishkin. Approximate parallel scheduling. Part II. Applications to logarithmic-time optimal parallel graph algorithms. *Information and Computation*, 92(1):1–47, May 1991.

[7] Richard John Cole and Uzi Vishkin. Approximate parallel scheduling. Part I: The basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM Journal on Computing*, 17(1):128–142, February 1988.

[8] Narsingh Deo, Amit Jain, and Muralidhar Medidi. An optimal parallel algorithm for merging using multiselection. *Information Processing Letters*, 50(2):81–88, 1994.

[9] G. N. Frederickson and D. B. Johnson. The complexity of selection and ranking in $X+Y$ and matrices with sorted columns. *Journal of Computer and System Sciences*, 24:197–208, 1982.

[10] G. N. Frederickson and D. B. Johnson. Finding kth paths and p-centers by generating and searching good data structures. *Journal of Algorithms*, 4(1):61–80, 1983.

[11] T. Hagerup and C. Rüb. Optimal merging and sorting on the EREW PRAM. *Information Processing Letters*, 33:181–185, December 1989.

[12] Joseph Ja′Ja′. *Introduction to Parallel Algorithms*. Addison Wesley, 1992.

[13] D. B. Johnson and T. Mizoguchi. Selecting the Kth element in $X+Y$ and $X_1+X_2+\ldots+X_m$. *SIAM Journal on Computing*, 7(2):147–153, May 1978.

[14] R. Karp and V. Ramachandran. Parallel algorithms for shared memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, Cambridge, MA, 1990. The MIT Press.

[15] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[16] W. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries on 2-3 trees. In *Proceedings of ICALP, 154*, pages 597–609, July 1983.

[17] Ramnath Sarnath and Xin He. Efficient parallel algorithms for selection and searching on sorted matrices. In *Proceedings of the International Parallel Processing Symposium*, pages 108–111, 1992.