Note
Optimal algorithms for generalized searching in sorted matrices

Hong Shen

School of Computing and Information Technology, Griffith University, Nathan, QLD 4111, Australia

Received January 1996; revised January 1997
Communicated by M. Nivat

Abstract

We present a set of optimal and asymptotically optimal sequential and parallel algorithms for the problem of searching on an \( m \times n \) sorted matrix in the general case when \( m \leq n \). Our two sequential algorithms have a time complexity of \( O(m \log(2n/m)) \) which is shown to be optimal. Our parallel algorithm runs in \( O(\log(\log m/\log \log m) \log(2n/m^{1-z})) \) time using \( m/\log(\log m/\log \log m) \) processors on a COMMON CRCW PRAM, where \( 0 \leq z < 1 \) is a monotonically decreasing function on \( m \), which is asymptotically work-optimal. The two sequential algorithms differ mainly in the ways of matrix partitioning: one uses row-searching and the other applies diagonal-searching. The parallel algorithm is based on some non-trivial matrix partitioning and processor allocation schemes. All the proposed algorithms can be easily generalized for searching on a set of sorted matrices.

Keywords: CRCW PRAM; Matrix search problem; Optimal algorithm; Processors; Sorted matrix; Time complexity; Work-optimal

1. Introduction

We say that a matrix is sorted if all elements in each row and column are sorted in non-decreasing (lexicographical) order, respectively. Order statistics, especially selection, on sorted matrices has received much attention [2, 7, 8, 11, 12, 15, 16, 18, 19] due to its important applications in many fields [8, 9, 13, 14, 17]. Closely related to selection is the problem of searching a sorted matrix for the occurrence of a given element (key), which we call the matrix search problem. This problem arises in many
applications such as image processing and computational biology, and hence, has attracted considerable attention [1, 3-6, 16].

It has been proven that searching on an $n \times n$ sorted matrix requires $\Omega(n)$ time [3]. Optimal sequential algorithms for this case exist in the literature [1, 5]. Work-optimal parallel algorithm for this case has been given in [16], that runs in $O(\log \log n)$ time on a COMMON CRCW PRAM. Other parallel algorithms have also been developed for searching on sorted matrices [4] and on matrices with sorted columns [10].

In this paper we study the problem of generalized searching in $m \times n$ sorted matrix, where $m \leq n$. Clearly for this problem, $\Theta(n)$ is no more a lower bound when $m = o(n)$, and hence simply applying the existing $n \times n$ matrix searching algorithms will not be able to reach optimum in this case. Neither can trivial generalization of the existing results by splitting $X$ into $\lceil n/m \rceil$ $m \times m$ submatrices and searching each submatrix individually reduce the total work to below $O(n)$ ($O(\lceil n/m \rceil m) = O(n)$). It seems that not much work has been reported on optimal solutions to the generalized matrix search problem in the case $m \leq n$.

The main contributions of this paper are the following:

- We propose two optimal sequential algorithms based on row-searching and diagonal-searching respectively, both running in $O(m \log(2n/m))$ time. We claim the optimality by showing that $\Omega(m \log(n/m))$ is a lower time bound for the matrix search problem in the general case when $m \leq n$.

- We present an asymptotically work-optimal parallel algorithm that runs in $O(\log(\log m / \log \log m) \log(2n/m^1-z))$ time using $m/\log(\log m / \log \log m)$ processors on a COMMON CRCW PRAM, where $0 \leq z < 1$ is a monotonically decreasing function $m$.

We present our optimal sequential algorithms in Section 2 and asymptotically optimal parallel algorithm in Section 3, and conclude the paper in Section 4 with some open problems for future research.

2. Optimal sequential matrix searching

Consider the problem of searching for a given element in an $m \times n$ sorted matrix in the general case $m \leq n$. A straightforward solution is to search $m$ rows one by one using binary search, which requires a total time of $O(m \log n)$ and is optimal only when $m = o(n)$. Another naive algorithm searches either $m \times m$ submatrices one by one employing the known optimal $n \times n$ matrix searching algorithms, yielding $O(n)$ time in total, which is optimal only when $m = \Theta(n)$. Neither of these are optimal "globally" for the general case $m \leq n$. In this section we present two algorithms running in time $O(m \log(2n/m))$ for the generalized matrix search problem. Our first algorithm is based on row-searching and has a simple structure. The second algorithm using diagonal-searching approach is slightly more sophisticated, but saves an additive factor in time complexity. We claim that both algorithms achieve optimality by showing the lower bound for the problem.
Throughout the paper we assume that $X$ is an $m \times n$ sorted matrix, $4 \leq m \leq n$, and $e$ the element to be searched for. When $m < 4$, simply applying the naive algorithm searching rows one by one will reach the optimum.

The basic idea behind both our algorithms is the following: searching proceeds in phases on some submatrices with reduced sizes, where in each phase a maximal number of elements which cannot be candidates for $e$ are discarded.

We lay $X$ in the Cartesian plane and let $X(0,0)$ (the smallest) be at the southwest corner and $X(m-1,n-1)$ (the largest) at the northeast corner. Our first algorithm works by repeatedly searching for a pivot element on the middle row of $X$ which splits $X$ into submatrices. The algorithm is given as the following procedure and runs by call $\text{Search-1}(e, X[(0,0),(m-1,n-1)])$:

Algorithm $\text{Search-1}(e, X[(r,c),(r',c')])$

{\text{/*Search for $e$'s occurrence in $m \times n$ sorted matrix $X[(r,c),(r',c')].$\*/}

0. $\tilde{m} = r'$ - $r$; $\tilde{n} = c' - c$;

if ($\tilde{m} < 4$) \&\& ($\tilde{n} < 4$) then \{Use binary search on rows/columns; Exit\};

1. Use binary search to find a pivot element $x_{r,j}$ on the middle row indexed $\tilde{r} = r + \tilde{m}/2$
   such that $x_{r,j} \leq e \leq x_{r,j+1}$;

2. if ($e - x_{r,j}$) \&\& ($e - x_{r,j+1}$) then \{e is found; Quit\};

3. if $e < x_{r,c}$ then $\text{Search-1}(e, X[(\tilde{r} + 1,c),(r',c')])$
   else if $e > x_{r,c}$ then $\text{Search-1}(e, X[(\tilde{r} + 1,c),(r',c')])$
   else \{Search in $X_{\text{SW}}$ and $X_{\text{SE}}$ submatrices of reduced size.\}

The correctness of the algorithm is established by the following Lemma.

**Lemma 1.** During each phase of recursion in algorithm $\text{Search-1}$ all the elements discarded cannot be candidates for $e$.

**Proof.** The lemma result's directly from a standard argument based on the following fact:

In each phase of recursion $X$ is divided into 4 submatrices according to the pivot element found in Step 3: $X_{\text{SW}} = X[(r,c),(\tilde{r},j)]$, $X_{\text{NW}} = X[(\tilde{r} + 1,c),(r,j)]$, $X_{\text{NE}} = X[(\tilde{r} + 1,j + 1),(r',c')]$ and $X_{\text{SE}} = X[(r,j + 1),(\tilde{r},c')]$. Clearly, $e \notin X_{\text{SW}}$ if $e > x_{r,j}$, and $e \notin X_{\text{NE}}$ if $e < x_{r,j+1}$. \(\square\)

Now we analyze the time complexity of the algorithm. Let $t(m,n)$ be the time complexity for searching on $X$. Clearly, the algorithm decomposes $t(m,n)$ into three parts required for finding $x_{m/2,j}$, searching on $X_{\text{NW}}$ and searching on $X_{\text{SE}}$. So we have the following recurrence:

\[
t(1,n) = O(\log n), \quad t(m,1) = O(\log m),
\]

\[
t(m,n) = t(m/2,j) + t(m/2,n - j) + O(\log n).
\]

(1)
It is easy to verify that $t(m, n)$ is maximized when $|X_{NW}| = |X_{SE}|$, that is, $j = n/2$. In this case $X$ is halved in both dimensions in each phase of recursion, so at the end there are $m$ remaining submatrices, all with dimension $1 \times n/m$, to be searched. Thus, we obtain the solution of Eq. (1) as follows and leave the detailed proof to the reader:

$$t(m, n) = O(2m \log(2n/m) - \log(n/4)) = O(m \log(2n/m)). \quad (2)$$

Our second algorithm splits $X$ in each phase via searching for a pivot on the main diagonal of the middle $m \times m$ submatrix, rather than on the middle row of $X$. The main diagonal of a matrix is drawn from its southwest corner to northeast corner. The algorithm is presented as follows:

Algorithm Search-2($e$, $X[(r, c), (r', c')]$)

{*Search for $e$'s occurrence in $\tilde{m} \times \tilde{n}$ sorted matrix $X[(r, c), (r', c')]$.*}

0. $\tilde{m} = r' - r$; $\tilde{n} = c' - c$;
   \hspace{1em} if $(\tilde{m} < 4) \vee (\tilde{n} < 4)$ then \{Use binary search on rows/columns; Exit\};
   \hspace{1em} if $m > n/2$ then \{Search-1($e$, $X[(r, c), (r', c + \delta)]$); Search-1($e$, $X[(r, c + \delta), (r', c')]$); Exit\};
1. Split $X$ into $\lceil \tilde{n}/\tilde{m} \rceil$ submatrices of dimensions $\tilde{m} \times \tilde{n}$ from west to east, where
   $\tilde{m} = r' - r$ and $\tilde{n} = c' - c$;

2. Use binary search to find a pivot element $x_{d, \Delta + d}$ on the main diagonal of the middle submatrix (the $\lceil \tilde{n}/2\tilde{m} \rceil$th) such that $x_{d, \Delta + d} \leq e \leq x_{d+1, \Delta + d+1}$;

3. Steps 2–5 are the same as Algorithm-1, with $x_{\tilde{r}, \tilde{j}}$ and $x_{\tilde{r}, \tilde{j} + 1}$ being replaced by $x_{d, \Delta + d}$ and $x_{d+1, \Delta + d+1}$, respectively, in all their context.
   \{*Search in 2 submatrices of reduced size.*\}

The correctness of the algorithm is implied by Lemma 1.

Similar to Search-1, with the time for searching for a pivot being replaced by $O(\log m)$, the time complexity of the algorithm follows the following recurrence:

$$t(1, n) = O(\log n), \quad t(m, 1) = O(\log m), \quad t(m, m) = m,$$

$$t(m, n) = t \left( m - d, \frac{n - m}{2} + d \right) + t \left( d, \frac{n + m}{2} - j \right) + O(\log m). \quad (3)$$

Clearly, $t(m, n)$ reaches maximum when $d = m/2$. In this case each phase of recursion halves both dimensions, so there are $m$ submatrices of dimension $1 \times n/m$ remaining at the end which will be searched for $e$. Thus, it is easy to show that the solution of Eq. (3) is

$$t(m, n) = O(m \log(2n/m) + m - \log(m/4)) = O(m \log(2n/m)). \quad (4)$$

We now show that $\Omega(m \log(n/m))$ is a lower bound on the time complexity for our search problem, and hence prove the optimality of both of the above algorithms.
Any pair of elements across different $X_i$ and $X_j$ are unordered.

Fig. 1. Off-diagonal slice $X_i$ in row $i$ for $0 < i < \frac{m}{2}$ in a 7 x 21 sorted matrix

**Lemma 2.** Given an $m \times n$ sorted matrix $X$, $m \leq n$, and element $e$, any algorithm searching for the occurrence of $e$ in $X$ requires $\Omega(m \log(n/m))$ time in the worst case.

**Proof.** Along the same line as in [3] for proving the lower bound $\Omega(n)$ for searching in an $n \times n$ square sorted matrix, we use the following argument for our proof.

Construct “off-diagonal” slice $X_i = \{X[i,(m-i-1)n/m], X[i,(m-i-1)n/m + 1], \ldots, X[i,(m-i)n/m - 1]\}$ in row $i$ of $X$, $0 \leq i \leq m - 1$, as depicted in Fig. 1. Clearly, $X_i$ contains a sorted sequence of $n/m$ elements.

We know that searching $X_i$ for $e$ in the worst case for large rank of $e$ requires $\log |X_i| = \log(n/m)$ comparisons and hence, $\Theta(\log(n/m))$ time for any $i$. Since $\forall x_i \in X_i$ and $\forall x_j \in X_j$ there is no order between $x_i$ and $x_j$ for $0 \leq i \neq j \leq m - 1$ (this can be easily seen from Fig. 1), searching for $e$ in $X_0 \cup X_1 \cup \cdots \cup X_{m-1}$ requires to search each individual $X_i$ for $e$, for $i = 0, 1, \ldots, m - 1$, which in the worst case takes $\Theta(m \log(n/m))$ time. The lemma follows immediately from the fact that $e$ may fall into any $X_i$ and hence searching $X$ contains searching $\bigcup_{i=0}^{m-1} X_i$.

We say that an algorithm is optimal if its time complexity matches the lower bound for the problem. By Eqs. (2) and (4) and Lemma 2, we obtain our first theorem:

**Theorem 1.** Searching on an $m \times n$ sorted matrix can be completed optimally in $O(m \log(2n/m))$ time by algorithms Search-1 and Search-2, where $m \leq n$.

3. Asymptotically optimal parallel matrix searching

Now, we consider the generalized matrix search problem in the parallel environment. For searching on an $n \times n$ sorted matrix $X$, an algorithm running in time $O(\log \log n)$ using $O(n/\log \log n)$ processors on a COMMON CRCW PRAM was given in [16]. For
the case, when $X$ is of $m \times n$ dimensions, $m \leq n$, naive approaches of binary search on $X$’s rows or columns immediately yield solutions with $O(\log n)$-time $O(m \log n)$-work or $O(\log m)$-time $O(n \log m)$-work respectively. Using the algorithm in [16] may result in a solution of $O(\log \log m)$ time and $O(n/\log \log m)$ processors, but its total work is still far from the optimal bound shown in the previous section when $m \ll n$. It seems that neither the naive approaches nor the algorithm in [16] can lead to work-optimal or near-optimal solutions for the generalized matrix search problem.

Here we present a new algorithm that runs in $O(\log (\log m/\log \log m) \log (2n/m^{1-z}))$ time using $m/\log(\log m/\log \log m)$ processors on the COMMON CRCW model, $0 \leq z < 1$. Our algorithm has an asymptotically optimal work when $m$ is large enough.

The basic idea behind our algorithm is in phases to partition $X$ into submatrices called cells and identify those active cells possibly containing $e$ (and discard all others). Assume that $X$ is divided into $uv$ submatrices by $u$ rows and $v$ columns. Obviously, examining whether $e$ occurs in $X$ is equivalent to examining whether $e$ exists in each submatrix for all submatrices. A sorted matrix can be uniquely identified by its southwest corner element $x_{\min}$ and northeast corner element $x_{\max}$. Clearly, $x_{\min}$ is the smallest element and $x_{\max}$ the largest element. We call these the two extreme elements of the matrix. The following lemma is essential for our algorithm.

**Lemma 3.** If $X$ is divided into $u \times v$ submatrices by $u$ equally distanced rows and $v$ equally distanced columns, then there are at the most $u + v$ submatrices that may contain $e$ as a non-extreme element.

**Proof.** We use a standard approach and draw a main diagonal to connect $x_{\min}$ to $x_{\max}$ in each submatrix. A submatrix may contain $e$ iff $x_{\min} \leq e \leq x_{\max}$. Clearly all these main diagonals lie in at most $u + v$ diagonals of $X$. Since each diagonal of $X$ is sorted from its southwest end to the northeast end, there are at most one pair of points $x_{i,j}$ and $x_{i+1,j+1}$ which overlaps with the submatrix’s extreme points on the diagonal such that $x_{i,j} < e < x_{i+1,j+1}$. Hence $e$ may be a non-extreme element in at most $u + v$ submatrices.

Our algorithm works by partitioning $X$ into $m^{1/2} \times m^{1/2-\varepsilon}$ cells of size $m^{1/2} \times nm^{\varepsilon}/m^{1/2}$ for any small constant $0 \leq \varepsilon < \frac{1}{2}$, and identify all active cells (at most $m^{1/2} + m^{1/2-\varepsilon}$). Repeat this partitioning process until the splitting factor on the vertical direction $(m^{1/2-\varepsilon})$ shrinks to 1. Finally, search the sorted arrays for $e$ in every active cells using binary search. We present the algorithm as follows:

**Algorithm CRCW-Search($e$, $X$, $m$, $n$);**

```
{Search for $e$ in $m \times n$ sorted matrix $X$ on COMMON CRCW PRAM, $m \leq n$.}
(1) if $m \leq 4$ then find $e$ by binary search on every row in parallel; /*Trivial case.*}
    CELLS_0 ← {X};
    $m_0, n_0 ← m, n$; $i ← 0$;
    /*CELLS_i consists of all active cells of size $m_i \times n_i$, each represented by its extreme elements, for the $(i + 1)$th phase of partitioning, $i = 0, 1, \ldots$*.*/
```
(2) while \(m^{2^{-i}-\epsilon} \geq 1\) do

2.1 \(i \leftarrow i + 1;\)

2.2 for every cell in \(CELLS_{i-1}\) do in parallel

Partition it into \(u, \times v, = m^{2^{-i}} \times m^{2^{-i}}\) cells of size \(m, \times n,\)

using \(u, \times v,\) processors, where \(m, = m_{i-1}/u, = m^{2^{-i}}\) and \(n, = n_{i-1}/v, = n^{2^{-i}};\)

Assign one processor to each extreme element;

2.3 for every pair of extreme elements \((x_{min}, x_{max})\) do in parallel

if \((e = x_{min}) \lor (e = x_{max})\) then \{e is found; QUIT\};

if \(x_{min} < e < x_{max}\) then mark the corresponding cell “active”;  

2.4 for all active cells do in parallel

\(CELLS_{i} \leftarrow \{\)active cells\};

(3) for every cell in \(CELLS_{i}\) do in parallel

3.1 Allocate a set of processors to the cell;

3.2 for every processor in the cell do in parallel

3.2.1 Partition the cell into equally sized groups of rows and assign

a group to each processor;

3.2.2 Search each row within each processor’s group for \(e\) using binary

search;

QUIT if found.

{*Each cell in \(CELLS_{i}\) consists of \(m^{2^{-i}}\) sorted arrays.*}

The correctness of the algorithm is implied by the fact that only active cells may
contain \(e\) and thus all the inactive cells can be discarded in each phase of partitioning.

Now we analyze the above algorithm. Clearly, the while-loop in CRCW-Search iterates
\(\log(1/\epsilon)\) times. The following lemmas are needed for our analysis:

**Lemma 4.** The total number of processors used in Step 2 the while-loop in CRCW-
Search is bounded by \(\lambda^{\log(1/\epsilon)}m^{1-\epsilon}\), where \(1 < \lambda = 1 + m^{-\epsilon} < 2\) and \(0 \leq \epsilon < 1/2\).

**Proof.** In the \(i\)th phase partitioning, since each cell in \(CELLS_{i-1}\) is partitioned into
\(m^{2^{-i}} \times m^{2^{-i}}\) new cells and among these cells there are at most \(m^{2^{-i}} + m^{2^{-i}}\) active
cells by Lemma 3, \(0 \leq \epsilon < 1/2\), the total number of active cells in \(CELLS_{i}\) is

\[
|CELLS_{0}| = 1,
\]

\[
|CELLS_{i}| = (m^{2^{-i}} + m^{2^{-i}-\epsilon})|CELLS_{i-1}| = \lambda^{i}m^{\sum_{j=1}^{i}2^{-j}}, \ i > 1,
\]  \(5\)

where \(\lambda = 1 + m^{-\epsilon}\).

In Step 2.2 there are \(m^{2^{-i}} \times m^{2^{-i}} = m^{2^{-i}+1-\epsilon}\) processors assigned to each cell in
\(CELLS_{i-1}\), making the total number of processors required for this step to be

\[
p_{i} = m^{2^{-i}} \times m^{2^{-i}+1-\epsilon} = m^{2^{-i}+1-\epsilon}|CELLS_{i-1}|
\]

\[
= (\lambda^{i-1}m^{\sum_{j=1}^{i}2^{-j}})m^{2^{-i}+1-\epsilon} = \lambda^{i-1}m^{1-\epsilon}.
\]  \(6\)
Clearly, $p_i$ is the total number of processors required for the $i$th iteration of the while-loop, since it is needed also for Step 2.3 and $|CELLS_i| \leq p_i$ processors, by Eqs. (5) and (6), are used in Step 2.4.

The number of processors required for Step 2 the while-loop is the maximum number of processors required for each iteration:

$$p = \max_i(p_i) = \lambda \log(1/\varepsilon) m^{1-\varepsilon}.$$  

(7)

Because we use a COMMON CRCW PRAM, Step 2.3 can be completed in $O(1)$ time. This is achieved by simply letting every processor write the result of its comparison to a shared variable $s$ with initial value 1 — write "1" if "=" and "0" if "\neq", so that at the end, we know that $e$ is found if $s = 1$ and not found otherwise. All other steps inside the while-loop of Step 2 can clearly be done in $O(1)$ time. So Step 2 requires a total time of $O(\log(1/\varepsilon))$. The value of $\varepsilon$ is chosen such that the total work of Step 2 is optimal:

$$\lambda \log(1/\varepsilon) m^{1-\varepsilon} \leq m/\log(1/\varepsilon).$$  

(8)

It is easy to verify that the following equation is a solution of this inequality:

$$\varepsilon = \log \log m/\log m.$$  

(9)

By Eq. (9), Step 2 iterates $i^* = \log(\log m/\log \log m)$ times, and when it terminates, since $\lambda = 1 + m^{-\varepsilon} = 1 + 1/\log m$, we have the number of cells in $CELLS_i$.

$$|CELLS_{i^*}| = \lambda^{i^*} m^{\sum_{j=1}^{i^*} 2^{-j}} = (1 + 1/\log m)^{i^*} m^{1-1/2^{i^*}}$$

$$\approx (1 + i^*/\log m) m^{1-\log \log m/\log m} \leq 2m/\log m,$$  

(10)

and the size $m_{i^*} \times n_{i^*}$ of each cell in $CELLS_{i^*}$.

$$m_{i^*} = m^{2^{-i^*}} = m^{\log \log m/\log m} = \log m,$$  

(11)

$$n_{i^*} = nm^{i^*/\varepsilon} / m^{\sum_{j=1}^{i^*} 2^{-j}} = n/m^{1-(i^*+1)/\varepsilon}$$

$$= 2n/m^{1-(i^*+1)\varepsilon - 1/\log m} = n/m^{1-z},$$  

(12)

where

$$0 \leq z = \left(\log \left(\frac{\log m}{\log \log m} + 1\right) \log m - 1\right) / \log m < 1.$$  

Clearly, the value of $z$ monotonically decreases on $m$.

Based on the above equations, we now claim

**Lemma 5.** Step 3 in CRCW-Search can be completed in time $O(\log(\log m/\log \log m) \log(2n/m^{1-z}))$ using $m/\log(\log m/\log \log m)$ processors, where $0 \leq z < 1$ is a monotonically decreasing function on $m$.  


Proof. We show the processor allocation scheme for Step 3. By Eq. (10) at the most \(2m/\log m\) active cells in total exist when entering Step 3. So in Step 3.1, we can allocate \(\log m/(2\log(\log m/\log \log m))\) processors to each active cell. By Eq. (11) there are \(\log m\) (sorted) rows in each cell, therefore, we shall assign a group of \(2\log(\log m/\log \log m)\) rows to one processor within the cell (Step 3.2.1). Since a row in each cell is of length \(2n/m^{1-z}\) by Eq. (12), binary search on it in Step 3.2.2 requires \(O(\log(2n/m^{1-z}))\) time. Thus, the total time required for Step 3 is \(O(\log(\log m/\log \log m) \log(2n/m^{1-z}))\).

Combining Steps 2 and 3, we immediately have the following theorem.

**Theorem 2.** Searching on an \(m \times n\) sorted matrix \((m \leq n)\) can be completed in \(O(\log(\log m/\log \log m) \log(2n/m^{1-z}))\) time using \(m/\log(\log m/\log \log m)\) processors on a COMMON CRCW PRAM, where \(0 \leq z < 1\) is a monotonically decreasing function on \(m\).

It is clear that our algorithm CRCW-Search is asymptotically work-optimal, as \(z \to 0\) when \(m \to \infty\).

4. Concluding remarks

We have proposed two optimal \(O(m \log(2n/m))\)-time sequential algorithms for the problem of searching an \(m \times n\) sorted matrix for element \(e\) that is optimal in the general case when \(m \leq n\), whereas, existing algorithms may lead to solutions to this problem which are optimal only for either \(m = \Theta(n)\) or \(m = o(n)\). The basic idea behind both algorithms is in phases to partition the matrix into some submatrices with reduced sizes, discard all those that cannot contain \(e\), and proceed by searching on each remaining submatrix. The partitioning technique used in the first algorithm is row-searching, and the one used in the second algorithm is diagonal-searching. We have also presented a parallel algorithm running in \(O(\log(\log m/\log \log m) \log(2n/m^{1-z}))\) time using \(m/\log(\log m/\log \log m)\) processors on a COMMON CRCW PRAM, \(0 \leq z < 1\) is a monotonically decreasing function on \(m\), which is asymptotically work-optimal. The parallel algorithm is developed by using non-trivial problem partitioning and processor allocation schemes.

All our algorithms can be employed for searching on a set of sorted matrices in a straightforward way by searching either on each matrix individually or on the combined sorted matrix in which all given matrices are padded along the main diagonal [16, 18].

We believe that the techniques we developed for problem partitioning and processor allocation may be applicable for solving other relevant problems on order statistics on partially ordered structures.
Below are some open problems for future research:

- Whether we can eliminate the \( m^2 \)-factor in the time complexity of our parallel algorithm and make the algorithm optimal for any \( m \).
- Design of work-optimal parallel algorithms on CREW and EREW PRAM.

Acknowledgements

The author wishes to thank Ralph Johan Back for his support during the course of this work while the author was visiting Åbo Akademi University. Thanks also to an anonymous referee for various constructive comments that have strengthened the results of this paper.

References