JOURNAL OF ALGORITHMS **23,** 386–400 (1997)
ARTICLE NO. AL960824

# More Efficient Parallel Totally Monotone Matrix Searching*

Phillip G. Bradford† and Rudolf Fleischer†

*Max-Planck-Institut für Informatik, Im Stadtwald, D-666123 Saarbrücken, Germany*

and

Michiel Smid‡

*Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg, D-39106 Magdeburg, Germany*

Received January 3, 1996

We give a parallel algorithm for computing all row minima in a totally monotone $n \times n$ matrix which is simpler and more work efficient than previous polylog-time algorithms. It runs in $O(\lg n \lg\lg n)$ time doing $O(n\sqrt{\lg n})$ work on a CRCW PRAM, in $O(\lg n(\lg\lg n)^2)$ time doing $O(n\sqrt{\lg n})$ work on a CREW PRAM, and in $O(\lg n\sqrt{\lg n \lg\lg n})$ time doing $O(n\sqrt{\lg n \lg\lg n})$ work on an EREW PRAM. Since finding the row minima of a totally monotone matrix has been shown to be fundamental in the efficient solution of a host of geometric and combinatorial problems, our algorithm leads directly to improved parallel solutions of many algorithms in terms of their work efficiency. 

## 1. INTRODUCTION

Let $M$ be an $m \times n$ matrix whose entries belong to some totally ordered set. The *row minima problem* is to find for each row $i \in \{1, \ldots, m\}$ the index $\min(i)$ of the column that contains the minimal element of row $i$.

* The authors were supported by the ESPRIT Basic Research Actions Program, under Contract 7141 (Project ALCOM II). The first author was also partially supported by NSF Grant CCR-9203942 while he was at Indiana University.

† E-mail: {bradford, rudolf}@mpi-sb.mpg.de.
‡ E-mail: michiel@isg.cs.uni-magdeburg.de.



0196-6774/97 $25.00

The *row maxima problem* is defined symmetrically. Throughout this paper, we assume that all entries of $M$ are distinct; otherwise, we could replace entry $M_{i,j}$ by the triple $(M_{i,j}, i, j)$ and use the lexicographical order on these triples. We also assume that each entry $M_{i,j}$ can be accessed in constant time.

Clearly, the row minima problem has time complexity $\Theta(mn)$. It turns out, however, that many problems can be reduced to the row minimal problem for matrices of a special form.

DEFINITION 1. An $m \times n$ matrix $M$ is *monotone* if $\min(i) \leq \min(j)$ for all $1 \leq i < j \leq m$.

Aggarwal *et al*. [1] proved that solving the row minima problem on a monotone $m \times n$ matrix has time complexity $\Theta(m \lg n)$. They also observed that in many applications an even more restricted type of matrices occurs.

DEFINITION 2. An $m \times n$ matrix $M$ is *totally monotone* if every $2 \times 2$ minor is monotone. That is, for all $1 \leq i < k \leq m$ and $1 \leq j < l \leq n$, if $M_{i,j} > M_{i,l}$ then $M_{k,j} > M_{k,l}$.

Many geometric and combinatorial problems, such as computing extremal inscribed or circumscribed $k$-gons [1], wire routing [1], or the matrix chain ordering problem [6], can be reduced to the row minima problem on totally monotone matrices. Therefore, the parallel algorithm for the latter problem, which we develop in this paper, leads directly to improved parallel solutions for many problems. We remark that in all these examples it is not necessary to compute the whole matrix in advance, which would need $\Theta(mn)$ time. Rather, as is true for many applications of this problem, we assume that in constant time we can compute *any* matrix element.

Consider the following example from [1]: Given a convex $n$-gon $P$ in the place with vertices $p_0, \dots, p_{n-1}$, find for each vertex $p_i$ its further neighbor in $P$. This problem can be solved by finding the row minima of the following totally monotone $n \times (2n - 1)$ matrix $M$ (see [1] for details):

If $i < j \leq i + n - 1$ then $M_{i,j} = dist(p_i, p_{((j-1) \bmod n)+1})$.

If $j \leq i$ then $M_{i,j} = j - i$.

If $j \geq i + n$ then $M_{i,j} = -1$.

It was shown in [1] that for $m \leq n$ the row minima problem on a totally monotone $m \times n$ matrix can be solved in asympotically optimal $O(n)$ time, and so can be the all-furthest-neighbors problem for a convex $n$-gon.

Having settled the sequential complexity of the problem asymptotically, researchers began designing parallel algorithms for the row minima problem on totally monotone matrices. Let us assume from now on that $m = n$.

Aggarwal and Park [2] showed how to solve the problem in $O(\lg n)$ time and $O(n \lg n)$ work on a CRCW PRAM. They also gave an $O(\lg^2 n / \lg \lg n)$ (resp. $O(n^{\varepsilon})$)) time and $O(n \lg n / \lg \lg n)$ (resp. $O(n)$) work algorithm for the CREW PRAM (for any $\varepsilon > 0$). As Raman and Vishkin [8] pointed out, the two latter algorithms work on an EREW PRAM as well. Atallah and Kosaraju [4] gave an EREW PRAM algorithm that runs in $O(\lg n)$ time and does $O(n \lg n)$ work.

Raman and Vishkin [8] designed optimal *randomized* algorithms which run with high probability in $O(\lg n)$ (resp. $O(\lg \lg n)$) time on an EREW (resp. CRCW) PRAM doing $O(n)$ work.

Until now, no *deterministic* algorithm was known that solves the row minima problem for a totally monotone $n \times n$ matrix in polylogarithmic time and $o(n \lg n / \lg \lg n)$ work. In this paper, we give such an algorithm which, on an EREW PRAM, improves the work of all previous algorithms [2, 4] by a factor of almost $\Theta(\sqrt{\lg n})$. Moreover, it is faster than Aggarwal and Park's algorithm in [2] when their algorithm's work is minimized. More precisely, we prove the following theorem.

THEOREM 3 (Main Theorem). *We can solve the row minima problem on $n \times n$ totally monotone matrices*

- *on a* CRCW PRAM *in $O(\lg n \lg \lg n)$ time and $O(n\sqrt{\lg n})$ work,*
- *on a* CREW PRAM *in $O(\lg n (\lg \lg n)^2)$ time and $O(n\sqrt{\lg n})$ work,*
- *on an* EREW PRAM *in $O(\lg n \sqrt{\lg n \lg \lg n})$ time and $(O(n\sqrt{\lg n \lg \lg n})$ work.*

*On the* CREW *and* EREW PRAM, *there is in fact a tradeoff between time and work, the other extreme being $O(\lg n \lg \lg n)$ time and $O(n \lg n)$ work (see Table 1).*

TABLE 1
Comparing the Most Efficient Deterministic Polylog Time Parallel Solutions to the Row Minima Problem on $n \times n$ Totally Monotone Matrices

| PRAM model | Work | Time | Source |
|---|---|---|---|
| CRCW | $O(n\sqrt{\lg n})$ | $O(\lg n \lg \lg n)$ | Our results |
| CREW | $O(n\sqrt{\lg n})$ | $O(\lg n (\lg \lg n)^2)$ | Our results |
| EREW | $O(n \lg n / \lg \lg n)$ | $O(\lg^2 n / \lg \lg n)$ | [2, 8] |
| | $O(n\sqrt{\lg n \lg \lg n})$ | $O(\lg n \sqrt{\lg n \lg \lg n})$ | Our results |
| | $O(n \lg n)$ | $O(\lg n)$ | [4] |

*Note*. The results in the third line were given in [2] for the CREW PRAM, but [8] observed that they also hold for the EREW PRAM.

The rest of this paper is organized as follows. In Section 2, we recall some results about totally monotone matrices and about sorting subroutines which we need. In Section 3, we first outline our algorithm, then give the main routines in more detail in Subsections 3.2 and 3.3, and finally put the pieces together in Subsection 3.4. We close with some remarks in Section 4.

## 2. PRELIMINARIES

We start by recalling some results from the literature. The next proposition and lemma are implicitly from [1]. Let $M$ be a totally monotone $m \times n$ matrix. The next proposition follows directly from Definition 2.

PROPOSITION 4. *For any two columns $a < b$, there exists a unique row $k \in \{0, \dots, m\}$ called the* change-over *of $a$ and $b$, such that $M_{i,a} < M_{i,b}$ for all $i \leq k$ and $M_{j,a} > M_{j,b}$ for all $j > k$.*

We say that column $b \in \{1, \dots, n\}$ is *useless* if it does not contain any row minima. Obviously, if $m < n$ then $M$ contains at least $n - m$ useless columns.

LEMMA 5. *If there exists columns $a$ and $c$ with $a < b < c$ and rows $i, j \in \{1, \dots, m\}$ with $j \leq i + 1$ such that $M_{i,a} < M_{i,b}$ and $M_{j,b} > M_{j,c}$, then column $b$ is useless. Moreover, $b$ is useless if either $M_{1,b} > M_{1,c}$ or $M_{m,a} < M_{m,b}$.*

*Proof.* Proposition 4 implies that column $b$ cannot obtain a row minimum above row $i$ or below row $j$. (See Fig. 1 for the case when $j < i$). ∎

Aggarwal *et al.* [1] used this in their optimal sequential algorithm for totally monotone $n \times n$ matrices which works as follows: Throw away all even rows, walk along the diagonal of the remaining matrix and eliminate $n/2$ useless columns, solve the row minima problem recursively on the now $n/2 \times n/2$ matrix, then reinsert the even rows and find their minima in time $O(n)$.

The next theorem shows that the last step of this algorithm can be done efficiently in parallel. However, identifying many useless columns seems to be a difficult task to do in parallel.

THEOREM 6 (Aggarwal and Park [2], see also [8]). *Let $M$ be a totally monotone $n \times n$ matrix, and assume we are given the two minima for every $r$th row of $M$. Then there is an* EREW PRAM *algorithm that computes the remaining row minima in $O(r + \lg n)$ time using $n/r$ processors.*

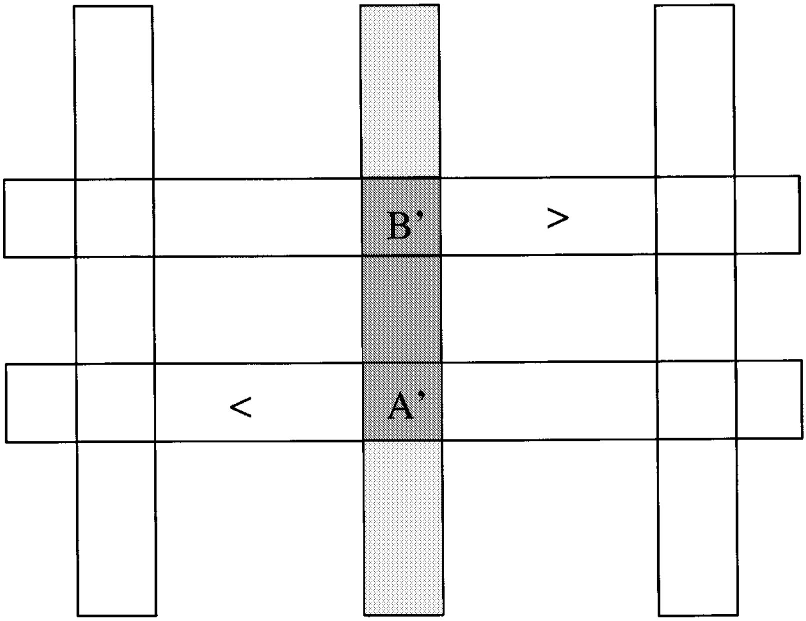


FIG. 1. The shaded column is useless because $A < A'$ and $B' > B$. The darkest portion of the shaded column contains elements known to be useless in two different ways.

Unfortunately, applying Theorem 6 recursively does not seem to give an efficient parallel algorithm. Therefore, we show how to identify useless columns efficiently in parallel.

Now, we give our basic approach for solving the row minima problem on totally monotone $n \times n$ matrices.

THEOREM 7 (Atallah and Kosaraju [4]). *Let $c \geq 1$ be some constant. Given an $n \times cn$ totally monotone matrix $M$, we can find its row minima in $O(\lg n)$ time using $n$ processors on an* EREW PRAM.

Atallah and Kosaraju [4] showed this for $c = 1$. So if we split $M$ into $c$ submatrices of size $n \times n$, we can find the row minima of each submatrix in time $O(\lg n)$ using $n$ processors, and then find the row minima of $M$ in time $O(1)$ by choosing between $c$ candidates in each row.

Let $M$ (resp. $N$) be an $m \times n$ (resp. $m \times n'$) matrix, where $n' \leq n$. We say that *$N$ has the same row minima as $M$*, if all columns of $M$ that contain row minima are also columns in $N$ and all of $N$'s columns are from $M$. Furthermore, $N$'s column ordering is directly inherited from the column ordering of $M$. That is, consider the useless columns in $M$ that by definition do not contain any row minima, then we can delete any number of these useless columns giving $N$ which "has the same" row minima as $M$.

In the next theorem, we will say that an algorithm computes an $m \times n'$ matrix $N$ that has the same row minima as a given $m \times n$ matrix $M$. This means that the matrix $N$ is represented implicitly in $O(m + n')$ space, that we can access every entry of $N$ in constant time, and that for each $1 \leq i \leq m$, if we are given the index of the column in $N$ that contains the minimal element of the $i$th row, then we can in constant time compute the index of the column in $M$ that contains the minimal element of the $i$th row.

THEOREM 8. *Let $c$ be a positive integer constant. Let $\mathscr{A}$ be a* PRAM *algorithm that, given a totally monotone $n/\lg n \times n$ matrix $M'$, computes a totally monotone $n/\lg n \times cn/\lg n$ matrix $M''$ that has the same row minima as $M'$. Let $f(n)$ (resp. $g(n)$) denote the amount of time (resp. work) this algorithm takes. Then the row minima problem on totally monotone $n \times n$ matrices can be solved in $O(\lg n + f(n))$ time and $O(n + g(n))$ work.*

*Proof.* Let $M$ be a totally monotone $n \times n$ matrix. Let $M'$ be the totally monotone $n/\lg n \times n$ matrix obtained by taking every $\lg n$th row of $M$. By our assumption, we can in $f(n)$ time and with $g(n)$ work compute a totally monotone $n/\lg n \times cn/\lg n$ matrix $M''$ that has the same row minima as $M'$. By Theorem 7, we can solve the row minima problem for $M''$ (and, hence, for $M'$) in $O(\lg n)$ time with $O(n)$ work. Given the row minima for $M'$, Theorem 6 implies that we can find all row minima of $M$ in $O(\lg n)$ time and $O(n)$ work. Note that the results of Theorems 6 and 7 hold for the EREW PRAM. As a result, we can solve the row minima problem for $M$ on the same PRAM model as that on which algorithm $\mathscr{A}$ works. ∎

Later in the proof of the main theorem we will apply this result with $c = 8$. So, for example, in the case of the CRCW PRAM we have $f(n) = O(\lg n \lg\lg n)$ and $g(n) = O(n\sqrt{\lg n})$ (See Theorem 17).

Theorem 8 implies that it suffices to design parallel algorithms that, given a totally monotone $n/\lg n \times n$ matrix $M'$, compute a totally monotone $n/\lg n \times cn/\lg n$ matrix $M''$, for some integer constant $c$, that has the same row minima. In the rest of this paper, we will show how to design such algorithms.

Note that the matrix $M''$ always exists: $M'$ has $n/\lg n$ rows and, hence, there are this many row minima. Hence, the main problem is to reduce the number of columns from $n$ to $cn/\lg n$.

Many of the efficient parallel algorithms for row minima searching use the strategy we adopt of finding the minima separately in blocks of columns. Then merging these minima gives the row minima of the original given matrix.

We note that Raman and Vishkin [8] used a similar strategy of ours in their randomized algorithm. In particular, given a totally monotone matrix they select appropriate rows via Theorem 6 and then use randomization to select columns. This gives a suitably smaller submatrix to which they eventually apply one of the less efficient algorithms giving (expected) polylog time and $O(n)$ work parallel algorithms.

The general strategy of taking blocks of columns and solving the row minima problem on them and then recombining these results is taken in Atallah and Kosaraju's [4] algorithm as well. Aggarwal and Park [2] gave an algorithm using this strategy costing $O(n \lg n)$ work. They also gave the first rendition of Theorem 6 for the CREW PRAM in [2] and Raman and Vishkin [8] pointed out that it holds for the EREW PRAM.

We close this section by mentioning some standard results for computing prefix sums and parallel integer sorting.

THEOREM 9. *Given $n$ 0/1-variables $x_1, \ldots, x_n$, we can compute all prefix sums $\sum_{i=1}^{k} x_i$, for $k = 1, \ldots, n$, on an* EREW PRAM *in $O(\lg n)$ time and $O(n)$ work.*

*Proof.* See for example [7]. ∎

THEOREM 10. *Given $n$ integer variables $x_1, \ldots, x_n \in \{1, \ldots, n\}$, we can stable sort them*

- *on a* CRCW PRAM *in $O((\lg n)/\lg\lg n)$ time and $O(n \lg\lg n)$ work;*
- *on a* CREW PRAM *in $O(\lg n \lg\lg n)$ time and $O(n\sqrt{\lg n})$ work;*
- *on an* EREW PRAM *in $O(\lg n \sqrt{\lg n/\lg\lg n})$ time and $O(n\sqrt{\lg n \lg\lg n})$ work.*

*On the* CREW *and* EREW PRAM, *there is in fact a tradeoff between time and work, the other extreme being $O(\lg n)$ time and $O(n \lg n)$ work.*

*Proof.* The CRCW algorithm is due to Bhatt *et al.* [5]. The CREW and EREW algorithms are due to Albers and Hagerup [3]. ∎

## 3. OUR ALGORITHM

### *3.1. The General Idea*

In this section we outline our algorithm for identifying many useless columns in an $r \times n$ totally monotone matrix $M$, where $r \ll n$ (later we will choose $r = n/\lg n$). Let $k = \sqrt{\lg r}$.

The algorithm runs in $l$ *phases*. (We will see later that we can choose $l = 3 \lg \lg n$.) When a new phase starts with an $r \times m$ matrix (where $m \geq 8r$), then during this phase we will identify and delete at least $\frac{1}{4}m$ useless columns, thus leaving a matrix of size at most $r \times \frac{3}{4}m$ for the next phase.

At the beginning of a phase, we partition the $r \times m$ matrix into blocks of $k$ contiguous columns (the last block may be smaller), and assign one processor to each block. Then each processor runs the procedure *Color_Block* independently on its block of columns. The phase ends with a run of *Color_All* on the entire $r \times m$ matrix. *Color_Block* tries to eliminate columns locally, whereas *Color_All* eliminates columns based on global information so that these columns may be far apart (*Color_Block* computes candidate columns which are potentially useless and turns them over to *Color_All*).

*Color_Block* uses three colors to color all columns: A *red* column is known to be useless, a *yellow* column still has a chance of being found useless in the procedure *Color_All*, and a *green* column will definitely survive this phase, but at the end of a phase there are at most two green columns in each block.

The yellow columns are always created pairwise, so we call such a pair a *yellow pair*. There is also an integer $row(a, b) \in \{1, \ldots, r\}$ attached to each yellow pair $(a, b)$, such that there exist columns $c$ and $d$, $a < c \leq d < b$, with $M_{row(a,b),a} > M_{row(a,b),c}$ and $M_{row(a,b),d} < M_{row(a,b),b}$; we call this property the *yellow-property* of a yellow pair. Further, all *row*-values within a block will be different.

*Color_All* will then work on the yellow pairs and find nearly as many useless columns as there are yellow pairs. It colors these useless columns red, the other columns green. The green columns can then be compacted into a smaller matrix which serves as input for the next phase. Without loss of generality, we assume that the columns are represented by an array of pointers $P$, with one pointer to each column. These columns can be compacted in several ways. For example, take an additional array $C$ of length $n$ with one array component for each column. For each red column make the corresponding array element contain 0 and for each green column make the corresponding array element contain 1. Now, apply a parallel prefix sum to $C$ giving $C'$. The $i$th column is to be deleted if

$C'[i-1] = C'[i]$, and define $C'[0] = 0$. So for each column $i$ such that $C'[i-1] \neq C'[i]$, move the pointer in $P[i]$ to $P[C'[i]]$. It is straightforward that this causes no write-conflicts. We will analyze the time and work of such compaction shortly.

### 3.2. *The Procedure Color_Block*

The input to the procedure *Color_Block* is an $r \times k$ matrix (recall that $k = \sqrt{\lg r}$). Since there is only one processor assigned to each block, it is a purely sequential algorithm.

At the beginning, all columns are colored green. Next *Color_Block* runs in *iterations*. In each iteration, we either throw away some columns or some rows. We remark that after each run of *Color_Block* all rows reappear. Further after each phase all columns that were not red reappear. We stop running iterations of *Color_Block* when only two rows are left.

COROLLARY 11. *Consider the top and bottom rows of a block. There is no local maximum* (*even after removing other red columns*) *other than the first and last elements of these rows.*

The proof follows directly from Lemma 5.

We maintain the following *iteration invariant*:

| |
|---|
| All columns are green, and the matrix elements in the top row are increasing from left to right, whereas the elements in the bottom row are decreasing. |

So if we have a matrix of two rows, we know from Corollary 11 that all columns except the first and the last ones are useless and can therefore be colored red.

We can easily guarantee the iteration invariant before the first iteration. We just scan through the first row, coloring all columns containing a local maximum red (these columns are useless by Lemma 5); this may include some backtracking, but each column is visited at most twice. Similarly, all columns right of a local minimum in the last row are useless and can be colored red.

Now, as well as after each iteration, we must deal with the columns we have just colored yellow or red. Since we need to delete these columns (at least conceptually), the easiest way seems to have two arrays *left* and *right* of size $k$ which contain for each green column its closest green neighbor to the left and to the right, respectively. Then each coloring (i.e., deletion) takes constant time. To make the algorithms simpler, we simulate two more columns 0 and $k + 1$ that are always green, and whose entries are all $\infty$. These two columns should obviously not be included in the iteration invariant.

*One Iteration*. Each iteration consists of two steps. Assume, the current matrix consists of rows $v, v + 1, \ldots, w - 1, w$ of our original $r \times k$ matrix.

First, we start a binary search for the changeover between the first two columns from row $v$ down to row $w$, but we stop after $k$ comparisons. This gives us two rows $i < j$ with $M_{i,1} < M_{i,2}$ and $M_{j,1} > M_{j,2}$. It $w - v = r'$, then $j - i = r'/2^k$.

Then we make rows $i$ and $j$ monotone by calling *ScanRow* for both of them (see Fig. 3). *ScanRow* $(s)$ first deletes useless columns until the elements of $s$ form a monotone decreasing chain followed by a monotone increasing chain. If the decreasing chain is not longer than the increasing chain, then we could pair all columns of the decreasing chain with columns of the increasing chain to create yellow pairs, except that then all of them would have the same *row*-value. Therefore, we call *ScanUp* which establishes a staircase of $>$'s as depicted in Fig. 2, and in the process eventually finds some more useless columns. Since all rows above $s$ must also be increasing where row $s$ is increasing (Proposition 4), we can now create yellow pairs with different *row*-values. Deleting them gives us a monotone increasing row $s$.

Similarly, we compute a downward staircase if the increasing chain is shorter; then row $s$ becomes decreasing. We note that *ScanUp* and

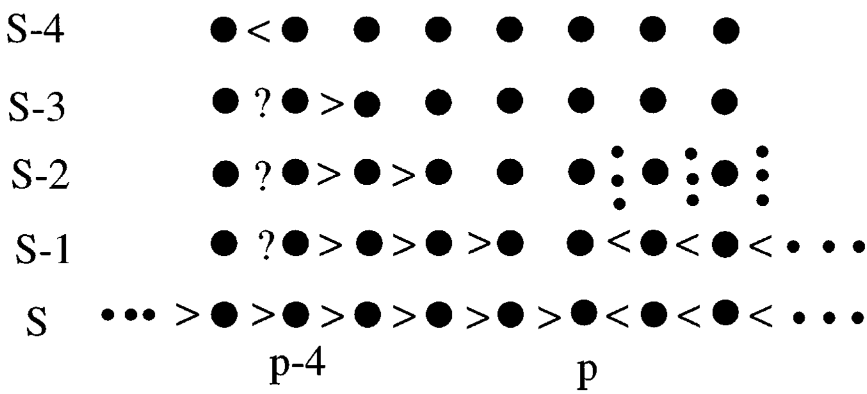


FIG. 2. Scanup starts at column $p$ and goes diagonally upwards (among the green columns) until it finds a "$<$" (here in row $s - 4$), then it deletes the right column $(p - 4)$ and backtracks to the row below (i.e., row $s - 3$). Here, we assume for simplicity that $left(x) = x - 1$ for all $x$.

```
Procedure ScanRow (s)
   -- Columns 0 and k + 1 are green, and M_{s,0} = M_{s,k+1} = ∞.
j = right(0);   -- first green column
while j < k + 1
do if (M_{s,left(j)} < M_{s,j}) and (M_{s,j} > M_{s,right(j)})
      then   j' = left(j);
             color column j red and delete it;   -- Lemma 5
             j = j',                             -- backtrack
      else   j = right(j);
   --Now the green columns form a pattern
   --   M_{s,1} > ··· > M_{s,p} < M_{s,p+1} < ··· < M_{s,q}
if p ≤ q/2 then   ScanUp (s, p);
                  for j = 1, ..., p − 1 do create yellow pairs (p − j, p + j)
                                           with row-value s − j + 1;
    else          ScanDown (s, p);
                  for j = 1, ..., q − p do create yellow pairs (p − j, p + j)
                                           with row-value s + j − 1;

   Procedure ScanUp (s, p)
      -- Search a diagonal of >'s going up from p.
      -- We know that M_{s,left(p)} > M_{s,p}
   p = left(p);   s = s − 1
   while p > 0      -- Invariant: M_{s+1,p} > M_{s+1,right(p)}
   do if M_{s,left(p)} > M_{s,p}
         then   s = s − 1;
                p = left(p);
         else   q = right(p);
                color p red and delete it;   -- Lemma 5
                s = s + 1;                   -- backtrack
                p = q;

  Procedure ScanDown (s, p)
     -- Search a diagonal of <'s going down from p.
     -- We know that M_{s,p} < M_{s,right(p)}.
  p = right(p);   s = s + 1
  while p < k + 1       -- Invariant: M_{s−1,left} < M_{s−1,p}
  do if M_{s,p} < M_{s,right(p)}
        then   s = s + 1;
               p = right(p);
        else   q = left(p);
               color p red and delete it;   -- Lemma 5
               s = s − 1;                   -- backtrack
               p = q;
```

FIG. 3. Procedures *ScanRow*, *ScanUp*, and *ScanDown*.

*ScanDown* are essentially the same as the procedure REDUCE used in the sequential algorithm by Aggarwal *et al*. [1] to find useless columns.

Among rows $\{v, i, j, w\}$, let $v'$ be the largest of the increasing rows, and $w'$ the smallest of the decreasing rows. By Proposition 4, $v' < w'$. Now we delete all rows above $v'$ and below $w'$, i.e., the next iteration works on rows $v', \dots, w'$. Clearly, the iteration invariant holds now.

LEMMA 12. *With the notation above we have*:

(a) *If no yellow pairs are created then* $v' = i$ *and* $w' = j$.

(b) *All yellow pairs have the yellow-property*.

(c) *The row-values of all yellow pairs are different*.

*Proof.* (a) If now yellow pairs are created, then procedures *ScanUp* and *ScanDown* have not been called, i.e., row $i$ is increasing and row $j$ is decreasing.

(b) By construction.

(c) Since the iteration invariant holds, *ScanUp* and *ScanDown* can never leave the submatrix on which the iteration started. Further, the rows which are used is *row*-value for yellow pairs do not belong to the submatrix of the next iteration. ∎

LEMMA 13. *After at most* $(3/2)k$ *iterations, the block consists of only two rows, i.e., the procedure Color_Block stops. The total time used for the iterations is* $O(\lg r)$.

*Proof.* In each iteration, we either find a yellow pair, or the $k$ probes of the binary search decrease the number of rows by a factor of $2^k$ (Lemma 12(a)). Since there can be no more than $k/2$ yellow pairs, after $(3/2)k$ iterations the number of rows would have shrunk to $r/(2^k)^k = 1$.

For the time bound, observe that *ScanRow* needs $O(k + s_i)$ time for iteration $i$, where $s_i$ is the number of red columns found by *Color_Block* in iteration $i$.

Since red columns are deleted once they are found, this sums to $O(k^2 + k) = O(\lg r)$ time for all iterations.

### *3.3. The Procedure Color_All*

The procedure *Color_All* takes all yellow columns and shows that at least a quarter of the yellow columns are useless. It works by sorting the yellow columns by their *row*-values and comparing them appropriately.

We may assume that *Color_Block* created exactly $k/2$ yellow pairs in each block, storing all of them in an array of size $m/2$ (we can add dummy yellow pairs which are later ignored). If we now sort the yellow pairs by their *row*-values, they will be grouped in contiguous blocks with the same *row*-value. The next lemma shows that now if two neighbors in the array happen to have the same *row*-value, then we can color one of the four columns involved red. This can easily be done with $O(u/\lg n)$ processors in $O(\lg n)$ time.

LEMMA 14. *Let* $(a, b)$ *and* $(s, t)$ *be two yellow pairs with* $b < s$. *If* $row(a, b) = row(s, t)$, *then either column* $b$ *or column* $s$ *is useless.*

*Proof.* Let $i = row(a, b) = row(s, t)$. The yellow-property and Lemma 5 imply that column $s$ is useless if $M_{i,b} < M_{i,s}$, otherwise column $b$ is useless. ∎

LEMMA 15. *Procedure Color_All runs*

- *on a* CRCW PRAM *in* $O((\lg m)/\lg\lg m)$ *time and* $O(m \lg\lg m)$ *work,*
- *on a* CREW PRAM *in* $O(\lg m \lg\lg m)$ *time and* $O(m\sqrt{\lg m})$ *work,*
- *on an* EREW PRAM *in* $O(\lg m \sqrt{\lg m/\lg\lg m})$ *time and* $O(m\sqrt{\lg m \lg\lg m})$ *work.*

*Proof.* Follows directly from Theorem 10. ∎

LEMMA 16. *If* $m \geq 8r$ *and* $k \geq 8$ $(i.e., n \geq 2^{64})$, *then at least* $m/4$ *columns will be colored red during Color_All.*

*Proof.* Assume that we have a total of $l_1$ red and $l_2$ yellow columns after running *Color_Block* on all blocks. Since there can be at most two green columns in each of the $m/k$ blocks, we have $l_1 + l_2 \geq m - (2m/k)$. (This includes the dummy yellow pairs that are also turned red.)

If there are $t_i$ yellow pairs with *row*-value $i$ then we will color $t_i - 1$ columns red (note that the sorting algorithm is stable, so the yellow pairs are ordered with increasing column numbers). Hence we will get a total of $l_1 + \Sigma_i(t_i - 1) = l_1 + 2/2 - r \geq m/2 - m/k - r \geq m/4$ red columns. ∎

### *3.4. Analysis of the Algorithm*

Let $r = n/\lg n$.

THEOREM 17. *After* $3\lg\lg n$ *phases of our algorithm, an* $r \times n$ *matrix* $M$ *is reduced to an at most* $r \times 8r$ *matrix with the same row minima as* $M$. *This takes*

- $O(\lg n \lg\lg n)$ *time and* $O(n\sqrt{\lg, n})$ *work on a* CRCW PRAM,
- $O(\lg n(\lg\lg n)^2)$ *time and* $O(n\sqrt{\lg n})$ *work on a* CREW PRAM,
- $O(\lg n \sqrt{\lg n \lg\lg n})$ *time and* $O(n\sqrt{\lg n \lg\lg n})$ *work on an* EREW PRAM.

*Proof.* After $l$ phases the matrix has at most $(3/4)^l n$ columns (Lemma 16). So after at most $3 \lg\lg n$ phases there are at most $8n/\lg n = 8r$ columns.

In phase $i$, *Color_Block* needs $O(\lg r) = O(\lg n)$ time and

$$0\left(\left(\tfrac{3}{4}\right)^i \frac{n}{k} \lg r\right) = O\left(\left(\tfrac{3}{4}\right)^i n \sqrt{\lg n}\right)$$

work (Lemma 13), so the total time for *Color_Block* over all phases is $O(\lg n \lg\lg n)$, and the total work is $O(n\sqrt{\lg n})$. The complexity bound now follows from Lemma 15. ∎

This together with Theorem 8 implies our Main Theorem which is:

MAIN THEOREM (Theorem 3). *We can solve the row minima problem on $n \times n$ totally monotone matrices*

- *on a* CRCW PRAM *in* $O(\lg n \lg\lg n)$ *time and* $O(n\sqrt{\lg n})$ *work,*
- *on a* CREW PRAM *in* $O(\lg n(\lg\lg n)^2)$ *time and* $O(n\sqrt{\lg n})$ *work,*
- *on an* EREW PRAM *in* $O(\lg n \sqrt{\lg n \lg\lg n})$ *time and* $O(n\sqrt{\lg n \lg\lg n})$ *work.*

*On the* CREW *and* EREW PRAM, *there is in fact a tradeoff between time and work, the other extreme being* $O(\lg n \lg\lg n)$ *time and* $O(n \lg n)$ *work.*

## 4. CONCLUSIONS

We have given an efficient deterministic parallel algorithm for computing the minima of all rows of a totally monotone matrix. For the CREW and EREW PRAM, the bottleneck is the sorting step.

But we do not really need sorting here, a weaker concept like *semi-sorting* [9] (i.e., group all equal elements together, not regarding the order between groups) would also suffice. Unfortunately, only efficient randomized algorithms are known for that problem.

Further, when we start a new phase of our algorithm in Section 2 we forget everything which we might have learned in previous phases. We cannot say exactly how much we lose here, but we have the impression that a thorough analysis could improve our complexity bounds.

Also, if we have a more general $m \times n$ matrix, then the cost will be based on a tradeoff of between the size of a block that *Color_Block* and *ScanRow* work on and the number of processors. Especially, since *ScanRow* sequentially runs across the blocks of columns.

If we have an $m \times n$ matrix, then the cost of the sorting step in *Color_All* depends on $n$ and not $m$. Therefore, in this case allowing $n$ to be asymptotically larger than $m$ will change our complexity bounds just as increasing the size of inputs to Albers and Hagerup's parallel sorting algorithm.

## ACKNOWLEDGMENTS

Thanks to Torben Hagerup for discussions about parallel sorting algorithms. Many thanks to the referees too.

## REFERENCES

1. A. Aggarwal, M. M. Klawe, S. Moran, P. Shor, and R. Wilber, Geometric applications of a matrix-searching algorithm, *Algorithmica* **2** (1987), 195–208.
2. A. Aggarwal and J. Park, Notes on searching multidimensional monotone arrays, *in* "Proceedings of the 29th Annual IEEE Symposium on the Foundations of Computer Science (FOCS'88), 1988," pp. 497–512. To appear in *J. Algorithms*.
3. S. Albers and T. Hagerup, Improved parallel integer sorting without concurrent writing, *in* "Proceedings of the 3rd Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'92), 1992," pp. 463–472.
4. M. J. Atallah and S. R. Kosaraju, An efficient parallel algorithm for the row minima of a totally monotone matrix, *J. Algorithms* **13** (1992), 394–413.
5. P. C. P. Bhatt, K. Diks, T. Hagerup, V. C. Prasad, T. Radzik, and S. Saxena, Improved deterministic parallel integer sorting, *Inform. and Comput.* **94** (1991), 29–47.
6. P. G. Bradford, G. J. E. Rawlins, and G. E. Shannon, Efficient matrix chain ordering in polylog time with linear processors, *in* "Proceedings 8th IEEE International Parallel Processing Symposium, 1994," pp. 234–241. To appear in *SIAM J. Comput*.
7. R. M. Karp and V. Ramachandran, Parallel algorithms for shared-memory machines, *in* "Handbook of Theoretical Computer Science, Vol. A," "Algorithms and Complexity," p. 875, Elsevier, New York, 1990.
8. R. Raman and U. Vishkin, Optimal randomized parallel algorithms for computing the row minima of totally monotone matrix, *in* "Proceedings 5th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'94), 1994," pp. 613–621.
9. L. G. Valiant, General purpose parallel architectures, *in* "Handbook of Theoretical Computer Science, Vol. A," "Algorithms and Complexity," p. 965, Elsevier, New York, 1990.