A Faster Parallel Algorithm for a Matrix Searching Problem

Mikhail J. Atallah

Abstract. We give an improved parallel algorithm for the problem of computing the tube minima of a totally monotone $n \times n \times n$ matrix, an important matrix searching problem that was formalized by Aggarwal and Park and has many applications. Our algorithm runs in $O(\log \log n)$ time with $O(n^2/\log \log n)$ processors in the CRCW-PRAM model, whereas the previous best ran in $O((\log \log n)^2)$ time with $O(n^2/(\log \log n)^2)$ processors, also in the CRCW-PRAM model. Thus we improve the speed without any deterioration in the $\text{time} \times \text{processors}$ product. Our improved bound immediately translates into improved CRCW-PRAM bounds for the numerous applications of this problem, including string editing, construction of Huffman codes and other coding trees, and many other combinatorial and geometric problems.

Key Words. Algorithms, Parallel computation, Monotone matrices.

1. Introduction. First we briefly review the problem, which was formalized by Aggarwal and Park [1]. We use somewhat different conventions than Aggarwal and Park.

Suppose we have an $n_1 \times n_2 \times n_3$ matrix $A$ and we wish to compute, for every $1 \leq i \leq n_1$ and $1 \leq j \leq n_3$, the $n_1 \times n_3$ matrix $\theta_A$ such that $\theta_A(i,j)$ is the smallest index $k$ that minimizes $A(i,k,j)$ (that is, among all $k$'s that minimize $A(i,k,j)$, $\theta_A(i,j)$ is the smallest index $k$ that minimizes $A(i,k,j)$ (that is, among all $k$'s that minimize $A(i,k,j)$, $\theta_A(i,j)$ is the smallest). We assume that $\theta_A$ satisfies the following sorted property:

$$\theta_A(i,j) \leq \theta_A(i, j + 1),$$

$$\theta_A(i,j) \leq \theta_A(i + 1, j),$$

and that, for every submatrix $A'$ of $A$, $\theta_{A'}$ also satisfies the above sorted property. All the $A$ matrices considered in this paper are assumed to satisfy the above two properties, which we call $\mathcal{P}$ (and these properties $\mathcal{P}$ are the only structure assumed by our algorithm).

---

1. This research was supported by the Office of Naval Research under Grants N00014-84-K-0502 and N00014-86-K-0689, the Air Force Office of Scientific Research under Grant AFOSR-90-0107, the National Science Foundation under Grant DCR-8451393, and the National Library of Medicine under Grant R01-LM05118. Part of the research was done while the author was at Princeton University, visiting the DIMACS center.

2. Department of Computer Science, Purdue University, West Lafayette, IN 47907, USA.

Received April 15, 1990; revised January 20, 1991. Communicated by F. Thomson Leighton.
The best CREW-PRAM algorithms for this problem run in $O(\log n)$ time and $O(n^2/\log n)$ processors [1], [3], and the best previous CRCW-PRAM algorithm ran in $O((\log \log n)^2)$ time and $O(n^2/(\log \log n)^2)$ processors [1] (where $n = n_1 = n_2 = n_3$).

The main result of this paper is a CRCW-PRAM algorithm of time complexity $O(\log \log n)$ and processor complexity $O(n^2/\log \log n)$. This improves on the speed of the previous $O((\log \log n)^2)$ bound, without any deterioration in the time $\times$ processors product.

We now review some concepts and definitions from [1] in order to relate the problem of computing $\theta_A$ to that work. Let $B$ be an $n_1 \times n_2 \times n_3$ matrix and, for each $1 \leq i \leq n_1$, let $\mu_2(i)$ and $\mu_3(i)$ denote the second and third coordinates of the minimum entry in the two-dimensional submatrix of $B$ consisting of those entries whose first coordinate is $i$ (if the submatrix contains multiple minima, we choose the first of these minima ordered lexicographically by their second and third component). Matrix $B$ is monotone [1] iff

1. For any $1 \leq i < i' \leq n_1$, we have $\mu_2(i) \leq \mu_2(i')$ and $\mu_3(i) \leq \mu_3(i')$.
2. For any $1 \leq i \leq n_1$, if we let $B'$ denote the two-dimensional submatrix of $B$ consisting of those entries whose first coordinate is $i$, then $B'$ satisfies the property that the minimum entry in its $j$th row is either below or to the right of the minimum entry in its $(j - 1)$st row. (If a row of $B'$ has several minima, then we choose the leftmost one.)

Matrix $B$ is totally monotone [1] iff every $2 \times 2 \times 2$ submatrix of $B$ is monotone. A totally monotone matrix is easily seen to satisfy the desired properties $\mathcal{P}$. The problem of computing $\theta_B$ for a totally monotone $B$ was called by Aggarwal and Park “computing the tube minima of a totally monotone matrix.” Thus the algorithm given in this paper in fact solves the tube minima problem. Many geometric and combinatorial applications of this problem are mentioned in [1].

There are applications to this problem in addition to those mentioned in [1]. Some of these are given in [5]: constructing Huffman codes in parallel, and other tree-construction problems. In [5] the problem was given the name “multiplying two concave matrices,” where a matrix $A$ is concave iff it satisfies the quadrangle inequality [10]: $A(i, j) + A(k, l) \leq A(i, l) + A(k, j)$ for all $i < k$ and $j < l$. In the framework of [5], we are given two $n \times n$ concave matrices $C$ and $D$ and we want to compute their $(\min, +)$ product $B = C \ast D$:

$$B(i, j) = \min_k (C(i, k) + D(k, j)).$$

If we let $A(i, k, j) = C(i, k) + D(k, j)$, then this reduces the problem of $(\min, +)$-multiplying $C$ and $D$ to that of computing the matrix $\theta_A$, where $\theta_A(i, j)$ is the smallest index $k$ for which $A(i, k, j)$ is minimized. The concavity of $C$ and $D$, and the fact that a submatrix of a concave matrix is itself concave, imply that $A$ satisfies the needed sorted property [1], [5].

Our improved CRCW-PRAM complexity bound for this problem immediately translates into improvements in the CRCW-PRAM complexities of its myriad
applications, some of which were mentioned above (we refrain from tabulating these, and refer the reader to [1] and [5] for some of the applications of this problem). We merely state our technical result:

**Theorem 1.** The $n \times n$ matrix $\theta_A$ of an $n \times n \times n$ matrix $A$ can be computed in $O(\log \log n)$ time with $O(n^2/\log \log n)$ processors in the CRCW-PRAM model.

Before going into the details, we point out that the main new ingredient in our approach consists of the judicious use of the "aspect ratio condition" (described later). All the other ingredients of the recipe that follows can be found in one form or another in [3] or, independently, in [1]. These ingredients are put together here in a different way from either [3] or [1]. The way we have put the pieces together to achieve the better bound is made possible by the aspect ratio condition: intuitively, the idea is to allow the aspect ratio of the subproblems solved recursively to deteriorate, but not too much, and in a controlled fashion. This is made precise later in the paper.

Although aspect ratios play a crucial role in establishing Theorem 1, once we have that theorem we can use it to solve problems of arbitrary aspect ratios (for more on this issue, see Section 4).

It will be convenient to analyze our algorithms using the time and work (i.e., number of operations) complexities. The processor complexity is deduced from these by using Brent's theorem [6], which states that any synchronous parallel algorithm taking time $T$ that consists of a total of $W$ operations can be simulated by $P$ processors in time $O(W/P + T)$. There are actually two qualifications to Brent's theorem before it can be applied to a PRAM:

(i) At the beginning of the $i$th parallel step, we must be able to compute the amount of work $W_i$ done by that step, in time $O(W_i/P)$ and with $P$ processors.
(ii) We must know how to assign each processor to its task.

Both qualifications (i) and (ii) to the theorem will be shown to be satisfied in our algorithms.

In Section 2 we give a preliminary algorithm for the case where $A$ is $l \times h \times l$ and $h \leq l^2$. That algorithm runs in $O(\log \log l)$ time and does $O((lh + l^2)(\log l)^2)$ work. Section 3 uses that preliminary algorithm to establish the above theorem. Section 4 makes further remarks, and Section 5 concludes.

Throughout, we refer to the first (resp. second, third) index of an entry of $A$ as its row (resp. column, height) index. Thus an $A$ that is $n_1 \times n_2 \times n_3$ has $n_1$ row indices, $n_2$ column indices, and $n_3$ height indices.

2. A Preliminary Algorithm. This section gives a preliminary algorithm that has the right time complexity, but does too much work (hence uses too many processors).

The procedure is recursive, and requires that $A$ be $l \times h \times l$ with $h \leq l^2$. We call this last condition the aspect ratio requirement; we assume it to be true initially, and we maintain it through the recursion (doing so without damaging the time
complexity or the work complexity is, in fact, the main difficulty in this preliminary algorithm). The preliminary CRCW-PRAM algorithm runs in $O(\log \log l)$ time and has work (that is, number of operations) complexity $O((lh + l^2)(\log l)^2)$.

Before describing the algorithm, we need a few definitions and a review of some properties.

Let $X$ (resp. $Y$) be a subset of the row (resp. height) indices of $A$, and let $Z$ be a contiguous interval of the column indices of $A$. The problem induced by the triplet $(X, Z, Y)$ is that of finding $\theta_A$ for the $|X| \times |Z| \times |Y|$ submatrix $A'$ of $A$ induced by $X$, $Z$, and $Y$. That is, it consists of finding, for each pair $u, v$ with $u \in X$ and $v \in Y$, the smallest index $k \in Z$ such that $A(u, k, v)$ is minimized. This $k$ need not equal $\theta_A(u, v)$ since we are minimizing only over $Z$. However, the following property holds [1], [3]. Assume $X$ (resp. $Y$) is a contiguous interval of row (resp. height) indices of $A$. Let $x, z, y$ (resp. $x', z', y'$) be the smallest (resp. largest) indices in $X, Z, Y$, respectively. If $\theta_A(x, y) = z$ and $\theta_A(x', y') = z'$, then the solution to the triplet $(X, Y, Z)$ gives the correct value of $\theta_A(u, v)$ for all $u \in X$ and $v \in Y$ (this follows from the sortedness of $\theta_A$).

The first stage of the computation partitions the row indices of $A$ into $l^{1/3}$ contiguous intervals $X_1, X_2, \ldots, X_{l^{1/3}}$ of size $l^{2/3}$ each. Similarly, the height indices of $A$ are partitioned into $l^{1/3}$ contiguous intervals $Y_1, Y_2, \ldots, Y_{l^{1/3}}$ of size $l^{2/3}$ each. An endpoint of an interval $X_i$ (resp. $Y_j$) is the largest or smallest index in it. For each pair $v, w$ such that $v$ is an endpoint of $X_i$ and $w$ is an endpoint of $Y_j$, we assign $h^{1 + (1/6)}$ processors which compute, in constant time, the index $\theta_A(v, w)$. (Computing the minimum of $h$ entries using $h^{1 + (1/6)}$ processors is known to take constant time [8].) The total number of processors used in this step of the algorithm is $O(l^{1/3} h^{1/3} h^{1 + (1/6)})$, which is $O(hl)$ because $h \leq l^2$.

Let $x$ (resp. $x'$) be the smallest (resp. largest) index in $X_i$, and let $y$ (resp. $y'$) be the smallest (resp. largest) index in $Y_j$. Let $Z_{i,j}$ be the interval $[a, b]$ where $a = \theta_A(x, y)$ and $b = \theta_A(x', y')$. In future, when we want to define such a $Z_{i,j}$, we shall simply say “let $Z_{i,j}$ denote the interval of column indices of $A$ defined by the set $\theta_A(v, w)$ such that $v \in X_i$ and $w \in Y_j$”; we do so for simplicity of expression, although it is an abuse of language (because $Z_{i,j}$ might include an index $k$ that is not the $\theta_A(u, v)$ of any pair $u \in X_i, v \in Y_j$).

After the above stage of the computation we know the beginning and end of each such interval $Z_{i,j}$. As already observed, for any pair of indices $u, v$ where $u \in X_i$ and $v \in Y_j$, we have $\theta_A(u, v) \in Z_{i,j}$. Thus it suffices to solve all of the subproblems defined by triplets $(X_i, Z_{i,j}, Y_j)$. However, some of these triplets might violate the aspect ratio condition because their $Z_{i,j}$ is too large (larger than $|X_i|^2 = l^{4/3}$): each such troublesome triplet (we call it a bad triplet) will be further partitioned into $k_{i,j} = \lceil |Z_{i,j}|/l^{4/3} \rceil$ smaller subproblems, by partitioning $Z_{i,j}$ into $k_{i,j}$ pieces of size $l^{4/3}$ each (except that possibly the $(k_{i,j})$th piece might be smaller). Specifically, if $Z_{i,j}^{(k)}$ denotes the $k$th piece from this partition of $Z_{i,j}$, then the $k$th subproblem spawned by the bad triplet $(X_i, Z_{i,j}, Y_j)$ is $(X_i, Z_{i,j}^{(k)}, Y_j)$. Of course such a spawned subproblem $(X_i, Z_{i,j}^{(k)}, Y_j)$ no longer has the property that $\theta_A(u, v) \in Z_{i,j}^{(k)}$ for $u \in X_i$ and $v \in Y_j$. However, the answer returned by solving such an $(X_i, Z_{i,j}^{(k)}, Y_j)$ is not meaningless: we can obtain $\theta_A(u, v)$ for $u \in X_i$ and $v \in Y_j$ by choosing the best among the $k_{i,j}$ candidates returned by the $k_{i,j}$ subproblems $(X_i, Z_{i,j}^{(k)}, Y_j)$,
1 \leq k \leq k_{i,j}. \text{ We are now ready to give the details of the second stage of the computation.}

The second stage of the computation “fills in the blanks” by doing one parallel recursive call on a number of problems, defined as follows. In what follows we describe these problems one at a time, but it should be kept in mind that they are all solved in parallel. The first class of problems to be solved recursively are the good ones, these defined by triplets \((X_i, Z_{i,j}, Y_j)\) where \(|Z_{i,j}| \leq l^{4/3}\). The \(Z_{i,j}\) of such a good problem is not large enough to violate the aspect ratio constraint (because it satisfies \(|Z_{i,j}| \leq |X_i|^2\)). The second class of problems to be solved recursively are those spawned by the bad triplets \((X_i, Z_{i,j}, Y_j)\), namely subproblems \((X_i, Z_{i,j}^{(k)}, Y_j)\), \(1 \leq k \leq k_{i,j}\). By definition, each such \((X_i, Z_{i,j}^{(k)}, Y_j)\) satisfies the aspect ratio requirement. When these recursive calls return, we need not do further work for the good triplets, but for the bad ones we only have the answers for the \(k_{i,j}\) subproblems they spawned. We can use these \(k_{i,j}\) subanswers to get the correct answers in constant time, however. For each bad triplet \((X_i, Z_{i,j}, Y_j)\), we need to compute, for every pair \(u \in X_i\) and \(v \in Y_j\), the minimum among \(k_{i,j}\) entries. We do so by using \(k_{i,j} h^{1/3}\) processors for each such pair \(u \in X_i\) and \(v \in Y_j\) (this is enough, since we are then computing the minimum of \(k_{i,j}\) entries using \(\geq k_{i,j}^{1+(1/6)}\) processors). Since there are \(l^{4/3}\) such \(u, v\) pairs per bad triplet, the total work done for this “bad triplet postprocessing” is upper-bounded by \(l^{4/3} h^{1/6} \sum_{i,j} k_{i,j}\); now, since

\[
\sum_{i,j} k_{i,j} = \sum_{\beta = -l^{4/3} + 1}^{l^{4/3} - 1} \sum_{i} k_{i,i+\beta} \leq \sum_{\beta = -l^{4/3} + 1}^{l^{4/3} - 1} (2h/l^{4/3}) = O(h/l),
\]

this work is \(O(l^{4/3} h^{1/6} h/l) = O(hl)\) (where the fact that \(h \leq l^2\) was used).

The bottom of the recursion is as usual: we stop when \(l\) is some small enough constant (note that, by the aspect ratio condition, a constant \(l\) implies a constant \(h\), since \(h \leq l^2\)).

The above description did not address the processor allocation problem: how processors are assigned, in constant time, to the subproblems they will solve recursively. We postpone discussing this issue until after we analyze the time and work complexities of the above algorithm (the machinery developed during that analysis will be used in the solution to the processor allocation problem).

**Analysis.** Before we analyze the complexity of the above algorithm, we make a straightforward observation that is needed in the analysis. Let \(\delta_{i,j}\) equal one if \(|Z_{i,j}| > l^{4/3}\), zero otherwise. Consider the sum

\[
H = \sum_{\delta_{i,j} = 0} |Z_{i,j}| + \sum_{\delta_{i,j} = 1} \sum_{1 \leq k \leq k_{i,j}} |Z_{i,j}^{(k)}|.
\]

This can be rewritten as follows, by changing the summation indices from \(i, j\) to \(i, \beta\):

\[
H = \sum_{\delta_{i,i+\beta} = 0} |Z_{i,i+\beta}| + \sum_{\delta_{i,i+\beta} = 1} \sum_{1 \leq k \leq k_{i,i+\beta}} |Z_{i,i+\beta}^{(k)}|.
\]
Let $H_\beta$ be the value of the above sum for a given value of $\beta$, that is, fixing $\beta$ and summing over $i$ (hence $H = \sum H_\beta$). It is not hard to see that $H_\beta$ is upper-bounded by $h + l^{1/3}$. Since there are $2l^{1/3} - 1$ possible choices for $\beta$, $H$ is upper-bounded by $2hl^{1/3} + 2l^{2/3}$. This fact is used in the analysis below.

The time and work complexities of the algorithm satisfy the recurrences:

$$T(l, h) \leq T(l^{2/3}, h') + c_1,$$
$$W(l, h) \leq c_2 lh + \sum_{\delta_{i,j} = 0} W(l^{2/3}, |Z_{i,j}|) + \sum_{\delta_{i,j} = 1} \sum_{1 \leq k \leq k_{i,j}} W(l^{2/3}, |Z_{i,j}(k)|),$$

where $c_1$ and $c_2$ are constants, and $h' \leq l^{4/3}$. The time recurrence clearly implies that $T(l, h) = O(\log \log l)$. We now prove, by induction on $l$, that the work recurrence implies that $W(l, h) \leq c(hl + l^2)(\log l)^2$ for a constant $c$. The basis of the induction is trivial. For the induction step, using the induction hypothesis in the above recurrence gives

$$W(l, h) \leq c_2 lh + \sum_{\delta_{i,j} = 0} c(l^{2/3}|Z_{i,j}| + l^{4/3})(\log(l^{2/3}))^2$$
$$ + \sum_{\delta_{i,j} = 1} \sum_{1 \leq k \leq k_{i,j}} c(l^{2/3}|Z_{i,j}(k)| + l^{4/3})(\log(l^{2/3}))^2.$$

Using the definition of $H$, the above inequality can be rewritten as

$$W(l, h) \leq c_2 lh + (4c/9)l^{2/3}(\log l)^2H + (4c/9)l^{4/3}(\log l)^2 \left( \sum_{\delta_{i,j} = 0} 1 + \sum_{\delta_{i,j} = 1} k_{i,j} \right).$$

Using the facts that $H \leq 2hl^{1/3} + 2l^{2/3}$ and that $\sum_{i,j} k_{i,j} \leq c'h/l$ for a constant $c'$, the above inequality implies

$$W(l, h) \leq c_2 lh + (8c/9)lh(\log l)^2 + (8c/9)l^{4/3}(\log l)^2$$
$$ + (4c/9)l^2(\log l)^2 + (4c/9)c'h(\log l)^2.$$

This clearly implies that $W(l, h) \leq c(hl + l^2)(\log l)^2$ for a suitably chosen constant $c$.

It should be clear from the above that a somewhat better bound for $W(l, h)$ can be obtained with a sharper analysis, but we choose not to bother with it; in fact, the rest of this paper can establish Theorem 1 even if we had a somewhat worse bound than the above one for $W(l, h)$, namely, $W(l, h) = O((lh + l^2)(\log l)^\alpha)$ is enough as long as $\alpha$ is constant (i.e., even if $\alpha > 2$). This will become clear in Section 3.

Processor Allocation. We now turn our attention to the issue of how the processors are allocated, in constant time, to the subproblems that must be solved recursively. The details of how this is done mimic the above analysis of the work complexity, and are somewhat tedious but not particularly difficult (they do involve some subtle points, however).
Imagine partitioning the subproblems to be solved recursively into classes, where class $\beta$ consists of the subproblems of the form $(X_i, Z_{i,i+\beta}, Y_{i+\beta})$ or $(X_i, Z_{i,i+\beta}^{(k)}, Y_{i+\beta})$. (Hence $-l^{1/3} + 1 \leq \beta \leq l^{1/3} - 1$.) The work to be done within class $\beta$ is

$$\text{Work}_{\beta} \leq c \left( \sum_{\delta_{i,i+\beta} = 0} \left( l^{2/3} |Z_{i,i+\beta}| + l^{4/3} \right) + \sum_{\delta_{i,i+\beta} = 1} \sum_{1 \leq k \leq k_{i,i+\beta}} \left( l^{2/3} |Z_{i,i+\beta}^{(k)}| + l^{4/3} \right) \right) (\log l)^2.$$ 

Rearranging the above and using the definition of $H_{\beta}$ gives

$$\text{Work}_{\beta} \leq c \left( l^{2/3} H_{\beta} + \sum_{\delta_{i,i+\beta} = 0} l^{4/3} + \sum_{\delta_{i,i+\beta} = 1} k_{i,i+\beta} l^{4/3} \right) (\log l)^2$$

$$= c \left( l^{2/3} H_{\beta} + l^{5/3} + 2h \right) (\log l)^2.$$

Therefore the number of processors we assign to class $\beta$ must be at least

$$(l^{2/3} H_{\beta} + l^{5/3} + 2h) (\log l)^2 (\log \log l)^{-1},$$

to within a constant factor (in fact, the constant factor can be taken to be unity, since we can trade a constant factor in the number of processors for a corresponding one in the time complexity). Since $H_{\beta} \leq h + l^{1/3}$, we can assign to class $\beta$ a number of processors equal to

$$\zeta(l, h) = (hl^{2/3} + l + l^{5/3} + 2h)(\log l)^2 (\log \log l)^{-1}.$$ 

This is easy to do in constant time, since the number of processors we assign to class $\beta$ does not depend on $\beta$. What remains to be shown is how, within class $\beta$, these $\zeta(l, h)$ processors are assigned to the various subproblems of that class. That is, in constant time, each subproblem $(X_i, Z_{i,i+\beta}, Y_{i+\beta})$ or $(X_i, Z_{i,i+\beta}^{(k)}, Y_{i+\beta})$ must be assigned the correct number of processors.

Within a class $\beta$, there is a natural left-to-right ordering of the subproblems: a subproblem is to the left of another one iff either

(i) its row interval is to the left of the other's row interval, or
(ii) in case they both have same row interval, its column interval is to the left of the other's column interval.

For example, subproblem $(X_i, Z_{i,i+\beta}^{(k)}, Y_{i+\beta})$ is to the left of subproblem $(X_{i'}, Z_{i',i'+\beta}^{(k')}, Y_{i'+\beta})$ iff either (i) $i < i'$ or (ii) $i = i'$ and $k < k'$. Two subproblems of class $\beta$ are neighbors iff one of them is immediately to the left of the other.

For a subproblem of the form $(X_i, Z_{i,i+\beta}, Y_{i+\beta})$, the number of processors to be assigned to it can be written as $f(l)|Z_{i,i}| + g(l)$ where $f(l) = l^{2/3} (\log l)^2 (\log \log l)^{-1}$ and $g(l) = l^{4/3} (\log l)^2 (\log \log l)^{-1}$. For a subproblem of the form $(X_i, Z_{i,i+\beta}^{(k)}, Y_{i+\beta})$, the number of processors to be assigned to it can be written as $f(l)|Z_{i,i+\beta}^{(k')}| + g(l)$. In either case we call the $g(l)$ portion fixed since it is the same for every subproblem, whereas the $f(l)|Z_{i,i+\beta}|$ or $f(l)|Z_{i,i+\beta}^{(k)}|$ portion is called variable.

In what follows we first discuss how to assign tentatively the variable portions, then we discuss the tentative assignment of the fixed portions (they are easier to
handle), and finally how to use these two tentative assignments to obtain the final assignment. In the final assignment we must be careful to make sure that the processors assigned to a particular subproblem form a contiguous interval of processor numbers (this is necessary because it is not enough for a set of processors to know that they are assigned to a certain subproblem; these processors must also have contiguous numbers).

Summed over all subproblems in class $\beta$, the total variable portion for class $\beta$ is $f(l)H_\beta$, and hence is at most $f(l)(h + l^{1/3})$ processors (because $H_\beta \leq h + l^{1/3}$). These $f(l)(h + l^{1/3})$ processors are partitioned into two groups, one of size $f(l)h$ (call it group $G_1$) and one of size $l^{1/3}f(l)$ (call it group $G_2$).

The $f(l)h$ processors of group $G_1$ are tentatively assigned as follows. Imagine associating, with each of the $h$ column indices, $f(l)$ processors, and view each column index as being the leader of the $f(l)$ processors assigned to it. Specifically, the $f(l)h$ processors of group $G_1$ are partitioned into $h$ chunks of size $f(l)$ each, and each chunk is associated with a column index (the $j$th chunk with the $j$th column index, which acts as its leader). We would like to assign all of the chunks whose leaders are in a column interval $Z_{i,i+\beta}$ or $Z_{i,i+\beta}^{(k)}$ to the subproblem corresponding to that interval, but there is a difficulty with this approach, in that a conflict can arise: a column index $j$ might belong to two neighboring subproblems, and we must then arbitrarily give the $j$th chunk to one of them and thus deprive the other subproblem. Observe that the only way for such a conflict over a column index $j$ to arise is when, for some $i, j$ is the right endpoint of $Z_{i,i+\beta}$ or $Z_{i,i+\beta}^{(k)}$, and is simultaneously the left endpoint of $Z_{i+1,i+1+\beta}$ or $Z_{i+1,i+1+\beta}^{(1)}$. In particular, no such conflict can arise between a $Z_{i,i+\beta}^{(k)}$ and the neighboring $Z_{i,i+\beta}$, since these two column intervals are (by definition) disjoint.

Now suppose that, whenever such a conflict occurs, we break the tie in favor of the leftmost of the two neighboring subproblems involved. Each subproblem that loses a conflict thus has a deficit of $f(l)$ processors. However, note that each such subproblem “with deficit” is of the form $(X_i, Z_{i,i+\beta}, Y_{i+\beta})$ or $(X_i, Z_{i,i+\beta}^{(1)}, Y_{i+\beta})$, $i > 1$; hence there are at most $l^{1/3}$ such subproblems, with a deficit of $f(l)$ processors each. We counteract these deficits by giving $f(l)$ of the processors of group $G_2$ to each subproblem of the form $(X_i, Z_{i,i+\beta}, Y_{i+\beta})$ or $(X_i, Z_{i,i+\beta}^{(1)}, Y_{i+\beta})$, $i > 1$ (note that some of these subproblems get the extra $f(l)$ processors without even having a deficit, i.e., even if they did not lose a conflict to their right neighbor).

Note that if a column does not belong to any column interval $Z_{i,i+\beta}$ or $Z_{i,i+\beta}^{(k)}$, then the chunk of processors associated with it simply remains unclaimed.

It is now easy to replace the above assignments for $G_1$ and $G_2$ with another assignment for $G_1 \cup G_2$ which, although also tentative, is better in that it assigns to each particular subproblem a variable portion consisting of processors having consecutive numbers. First, observe that the subset of that variable portion coming from group $G_1$ is already contiguous, and so is the subset coming from group $G_2$. It clearly suffices to show that a subproblem also knows the ranks, in each of $G_1$ and $G_2$, of the first processor assigned to it. That this holds for $G_2$ is trivial: the rank $v_2$ for a subproblem of the form $(X_i, Z_{i,i+\beta}, Y_{i+\beta})$ or $(X_i, Z_{i,i+\beta}^{(1)}, Y_{i+\beta})$, $i > 1$, is simply $(i - 2)f(l) + 1$. For $G_1$, if we let $j$ be the smallest column index in a subproblem’s column interval, then the rank is $v_1 = (j - 1)f(l) + 1$. 
This completes the tentative assignment of the variable portion of the processors needed by each subproblem. We now turn to the problem of tentatively assigning the fixed portion, that is, assigning \( g(l) \) processors to each subproblem.

Subproblems having a column interval of the form \( Z_{i,i+\beta} \) or \( Z_{i,i+\beta}^{(k)} \) are the easiest to handle, by assigning \( g(l) \) processors to every index \( i, 1 \leq i \leq l^{1/3} \): if \((X_i, Z_{i,i+\beta}, Y_{i+\beta})\) is a good triplet, then the \( g(l) \) processors for index \( i \) get assigned to it, otherwise (if it is a bad triplet) they get assigned to the \( k_{i,i+\beta} \)th subproblem spawned by that bad triplet.

For subproblems having a column interval of the form \( Z_{i,i+\beta}^{(k)} \) where \( k < k_{i,i+\beta} \), we exploit that fact that \( |Z_{i,i+\beta}^{(k)}| = l^{4/3} \), as follows. Imagine marking every column index that is a multiple of \(( h/l^{4/3}) \) as being "special," and associating with each special column \( g(l) \) processors. Since \( |Z_{i,i+\beta}^{(k)}| = l^{4/3} \), column interval \( Z_{i,i+\beta}^{(k)} \) contains exactly one special column. Detecting this column is trivial to do in constant time (it is the unique multiple of \( l^{4/3} \) within column interval \( Z_{i,i+\beta}^{(k)} \)). This completes the tentative assignment of the fixed portion of the processors needed by each subproblem.

We now discuss how the final assignment of processors is obtained from the above two tentative ones. Let \( G \) be the group of processors assigned to class \( \beta \), and let \( G' \) (resp. \( G'' \)) be the total fixed (resp. variable) portion of \( G \) (hence \( G = G' \cup G'' \)). Since each subproblem knows, as a by-product of the above tentative assignments, the rank within \( G' \) (resp. \( G'' \)) of the first processor assigned to it in \( G' \) (resp. \( G'' \)), it can easily find out the rank of the first processor to be assigned to it in \( G \). Once that rank is known, the processors that were tentatively assigned to the subproblem from \( G' \) and \( G'' \) can "mark," in constant time, a contiguous interval of processors in \( G \) that get assigned to that subproblem. This concludes the discussion of the processor assignment issue.

3. Decreasing the Work Done. Let us go back to the original goal of computing the \( \theta_A \) matrix for an \( n \times n \times n \) matrix \( A \), in \( O(\log \log n) \) time and \( O(n^2/\log \log n) \) processors.

Let \( ALGO_0 \) denote the algorithm of the previous section (recall that it has the right time complexity but does a factor of \(( \log n)^2\) too much work). There is more than one way to decrease the work done. The way we do it in this section has the advantage of being self-contained (in Section 4 we sketch another way, one that uses as a subroutine the CREW-PRAM Algorithm of [1] or [3]).

Using algorithm \( ALGO_0 \), we create an algorithm \( ALGO_1 \) that runs in \( O(\log \log n) \) time with \( O(n^2(\log \log n)^2) \) work. Then, using \( ALGO_1 \), we create an algorithm \( ALGO_2 \) that runs in \( O(\log \log n) \) time with \( O(n^2(\log \log \log n)^2) \) work. Finally, using \( ALGO_2 \), we create an algorithm \( ALGO_3 \) that runs in \( O(\log \log n) \) time with \( O(n^2) \) work.

The method for obtaining \( ALGO_k \) from \( ALGO_{k-1} \) is similar for \( k = 1, 2, 3 \), and uses the following lemma.

**Lemma 1.** Let \( ALGO' \) and \( ALGO'' \) be two algorithms for computing \( \theta_A \), running in time (respectively) \( T'(n) \) and \( T''(n) \), and doing work (respectively) \( W'(n) \) and \( W''(n) \). Then, for any \( 1 \leq s \leq n \), we can construct a third algorithm for computing
\( \theta_A \) that runs in time \( O(T'(n/s) + T''(s) + \log \log s + \log \log (n/s)) \) and does work \\
\( O(sW'(n/s) + (n/s)^2W''(s) + n^2) \).

**Proof.** We give an algorithm that makes use of \( ALGO' \) and \( ALGO'' \). The row indices of \( A \) get partitioned into \( n/s \) intervals \( X_1, \ldots, X_{n/s} \) of length \( s \) each. The height indices of \( A \) get partitioned into \( n/s \) intervals \( Y_1, \ldots, Y_{n/s} \) of length \( s \) each. The column indices of \( A \) get partitioned into \( s \) intervals \( Z_1, \ldots, Z_s \) of length \( n/s \) each. Let \( E_X \) (resp. \( E_Y \)) be the set of \( 2(n/s) \) endpoints of the \( X_i \)'s (resp. \( Y_i \)'s). Then we do the following:

1. We run, in parallel, \( s \) copies of \( ALGO' \), one on each of the \( s \) triplets \\
\( (E_X, Z_1, E_Y), \ldots, (E_X, Z_s, E_Y) \). This takes time \( T'(n/s) \) and work \( sW'(n/s) \).

2. For each \( u \in E_X \) and \( v \in E_Y \), we compute the correct \( \theta_A(u, v) \) value by taking the best among the \( s \) answers for the pair \( u, v \) returned by the solutions to the \( s \) triplets of the previous stage. We do so in \( O(\log \log s) \) time and \( O(s) \) work for each such pair \( u, v \). Since there are \( O((n/s)^2) \) such pairs \( u, v \), the total work done is \( O(n^2/s) = O(n^2) \). If we let \( Z_{i,j} \) denote the interval of column indices of \( A \) defined by the set \( \theta_A(v, w) \) such that \( v \in X_i \) and \( w \in Y_j \), then after this stage of the computation we know the beginning and end of each such interval \( Z_{i,j} \).

3. For every \( Z_{i,j} \) such that \( |Z_{i,j}| \leq s \), we solve the triplet \( (X_i, Z_{i,j}, Y_j) \) by using \( ALGO'' \). However, algorithm \( ALGO'' \) assumes unit aspect ratio ("square" matrices), whereas here we might have \( |Z_{i,j}| < |X_i| \). We get around this problem simply by making the matrices square (padding with dummy \( +\infty \) entries that cannot alter the correctness of the answer returned). Of course this means that we now do \( W''(s) \) work for each such triplet. However, since there are at most \( (n/s)^2 \) such triplets, the total work for this stage is \( (n/s)^2W''(s) \). The time is, of course, \( T''(s) \).

4. For every \( Z_{i,j} \) such that \( |Z_{i,j}| > s \), we partition \( Z_{i,j} \) into \( k_{i,j} = \lceil |Z_{i,j}|/s \rceil \) intervals \( Z_{i,j}^{(1)}, Z_{i,j}^{(2)}, \ldots, Z_{i,j}^{(k_{i,j})} \), of size \( s \) each (except that \( Z_{i,j}^{(k_{i,j})} \) might be smaller). Then we solve each triplet \( (X_i, Z_{i,j}^{(k)}, Y_j) \) by using \( ALGO'' \) (if \( k = k_{i,j} \), then we might need to "pad" the matrix in order to make it square, as in the previous stage). The time is \( T''(s) \), and the work per subproblem is \( W''(s) \). Since there are at most \( \sum_{i,j} k_{i,j} \) such subproblems, and since \( \sum_{i,j} k_{i,j} = O((n/s)^2) \), the total work for this stage is \( O((n/s)^2W''(s)) \).

5. For every \( Z_{i,j} \) such that \( |Z_{i,j}| > s \), we compute the right answer for each pair \( u \in X_i \) and \( v \in Y_j \), from among the \( k_{i,j} \) possibilities available from the previous stage. We do this in \( O(\log \log k_{i,j}) = O(\log \log (n/s)) \) time and \( O(k_{i,j}) \) work for each such pair \( u, v \). Since there are \( s^2 \) such pairs \( u, v \) for each such \( Z_{i,j} \), the total work for this stage is \( O(s^2 \sum_{i,j} k_{i,j}) = O(s^2(n/s)^2) = O(n^2) \).

It is clear that the above procedure proves the lemma.

\[ \square \]

To obtain \( ALGO_1 \), we use Lemma 1 with \( s = (\log n)^2 \) and with \( ALGO' = ALGO'' = ALGO_0 \).

To obtain \( ALGO_2 \), we use Lemma 1 with \( s = (\log \log n)^2 \) and with \( ALGO' = ALGO'' = ALGO_1 \).

To obtain \( ALGO_3 \), we use Lemma 1 with \( s = (\log \log \log n)^2 \), with \( ALGO' = ALGO_2 \), and using for \( ALGO'' \) the known optimal sequential algorithm for this
problem [1], [3] (which takes quadratic time, hence both \(W'(s)\) and \(T'(s)\) are \(O(s^2)\)).

Brent’s theorem [6] then implies an \(O(n^2/\log \log n)\) processor bound for \(ALGO_3\), thus establishing Theorem 1. We do not give the details of the processor allocation schemes, since they are very similar to those in Section 2—in fact, here we could even afford to assign processors in \(O(\log \log n)\) time rather than in constant time, since the above scheme for obtaining \(ALGO_k\) from \(ALGO_{k-1}\), \(1 \leq k \leq 3\), did not involve any recursive calls to \(ALGO_k\).

4. Further Remarks. Using algorithm \(ALGO_3\), we can tackle problems having different aspect ratios from those considered so far. By way of example, suppose \(A\) is \(l \times h \times l\) where \(h > l^2\), i.e., the aspect ratio condition is violated. For that case we can get an \(O(\log \log l + \log \log (h/l))\) time, \(O(lh)\) work algorithm as follows. Let \(X\) (resp. \(Y\)) be the set of all row (resp. height) indices of \(A\). Partition the column indices of \(A\) into \(q = \lceil h/l \rceil\) intervals of size \(l\) each (the \(q\)th interval may be smaller). Let these \(q\) intervals be \(Z_1, \ldots, Z_q\). Use \(q\) copies of \(ALGO_3\) to solve in parallel all the \((X, Z_i, Y)\) triplets. This takes \(O(\log \log l)\) time and \(O(ql^2) = O(lh)\) work. Then, for each pair \(u \in X\) and \(v \in Y\), we assign \(q\) processors to compute the correct \(\theta_A(u, v)\) in \(O(\log \log q)\) time (this involves taking the min of \(q\) quantities). The total work for this “postprocessing” is \(O(l^2q) = O(lh)\), and the time is \(O(\log \log (h/l))\).

We also note that even if our \(ALGO_6\) had done a polylog factor more work than the \(ALGO_6\) we gave in Section 2, we would still have been able to design an \(ALGO_k\) that runs in \(O(\log \log n)\) time and does only quadratic work. We would simply have had to use Lemma 1 a few more times, and end up with an \(ALGO_k\) with \(k > 3\).

An alternative method of decreasing the work done was suggested to us by Professor L. L. Larmore [7], after he read a draft of this paper. The idea is to use Lemma 1 with \(ALGO' = ALGO_6\) and with \(ALGO'' = any of the known optimal CREW-PRAM algorithms for that problem [1], [3].\)

5. Conclusion. We gave an \(O(\log \log n)\) time, \(O(n^2/\log \log n)\) processor algorithm for computing the tube minima of an \(n \times n \times n\) totally monotone matrix in the CRCW-PRAM model. As observed by Professor L. L. Larmore [7], these bounds can easily be shown to be optimal among quadratic-work algorithms for this model (this follows from Valiant’s \(\Omega(\log \log n)\) lower bound for computing the minimum of \(n\) entries with \(O(n)\) work [9]). Aggarwal and Park [1] introduced other versions of such parallel matrix searching problems. Optimal (or improved) algorithms for other such problems would be of great interest, especially in view of the wide applicability of matrix searching problems. In this respect, improved bounds for computing the row minima of an \(n \times n\) totally monotone matrix were recently discovered [4], achieving \(O(\log n)\) time and \(O(n \log n)\) work in the EREW-PRAM model, as opposed to the previous \(O(\log n \log \log n)\) time and \(O(n \log n)\) work in the stronger CREW-PRAM model. Another interesting avenue of research is the parallel solution of these problems on more realistic models than the PRAM, such as hypercubes (see, e.g., [2]).
Acknowledgments. The author is grateful to Professor L. L. Larmore, Dr. J. K. Park, and an anonymous referee for their comments on an earlier version of this paper.

References


