1990

An Efficient Parallel Algorithm for the Row Minima of a Totally Monotone Matrix

Mikhail J. Atallah
Purdue University, mja@cs.purdue.edu

S. Rao Kosaraju

Report Number:
90-959

AN EFFICIENT PARALLEL ALGORITHM FOR THE ROW MINIMA OF A TOTALLY MONOTONE MATRIX

Mikhail J. Atallah
S. Rao Kosaraju

CSD-TR-959
February 1990
(Revised April 1991)

An Efficient Parallel Algorithm for the Row Minima of a Totally Monotone Matrix

Mikhail J. Atallah*
Purdue University

S. Rao Kosaraju†
Johns Hopkins University

Abstract

We give a parallel algorithm for the problem of computing the row minima of a totally monotone two-dimensional matrix. Whereas the previous best CREW-PRAM algorithm for this problem ran in $O(\log n \log\log n)$ time with $O(n/\log\log n)$ processors, our algorithm runs in $O(\log n)$ time with $O(n)$ processors in the (weaker) EREW-PRAM model. Thus we simultaneously improve the time complexity without any deterioration in the *time* $\times$ *processors* product, even using a weaker model of parallel computation.

*Dept. of Computer Science, Purdue University, West Lafayette, IN 47907. This author's research was supported by the Office of Naval Research under Contracts 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.

†Dept. of Computer Science, Johns Hopkins University, Baltimore, MD 21218. This author's research was supported by the National Science Foundation under Grant CCR-8804284 and NSF/DARPA Grant CCR-8908092.

1 Introduction

First we review the problem, which was introduced and solved in linear sequential time in [2]. It has myriads of applications to geometric and combinatorial problems [1, 2].

For any $m \times n$ matrix A, the row minima are an m-vector θ_A such that, for every row index r ($1 \leq r \leq m$), $\theta_A(r)$ is the smallest column index c that minimizes $A(r,c)$ (that is, among all c's that minimize $A(r,c)$, $\theta_A(r)$ is the smallest). If θ_A satisfies the following *sorted* property:

$$\theta_A(r) \leq \theta_A(r+1),$$

and if for every submatrix A' of A, $\theta_{A'}$ also satisfies the sorted property, then matrix A is said to be *totally monotone* [1, 2]. (Note: this definition may appear different from that given in [1, 2], but it is easily seen to be equivalent, by negating all the entries of the matrix.)

Given a totally monotone matrix A, the problem of computing the θ_A array is known as that of "computing the row minima of a totally monotone matrix" [1]. The best previous CREW-PRAM algorithm for this problem ran in $O(\log n \log\log n)$ time and $O(n/\log\log n)$ processors [1] (where $m = n$). The main result of our paper is an EREW-PRAM algorithm of time complexity $O(\log n)$ and processor complexity $O(n)$. This improves on the time complexity of [1] without any increase in the *time* $\times$ *processors* product, and using a weaker parallel model. It also implies corresponding improvements on the parallel complexities of the many applications of this problem (which we refrain from listing—they can be found in [1, 2]). More specifically, our bounds are as follows.

Theorem 1.1 *The row minima (that is, the array θ_A) of an $m \times n$ totally monotone matrix A can be computed in $O(\log m + \log n)$ time with $O(m+n)$ processors in the EREW-PRAM model.*

In fact we prove a somewhat stronger result: that an implicit description of θ_A can be computed, within the same time bound as in the above theorem, by $O(n)$ processors. From this implicit description, a single processor can obtain any particular $\theta_A(r)$ value in $O(\log n)$ time.

Section 3 gives a preliminary result that is a crucial ingredient of the scheme of Section 4: an $O(\max(n, \log m))$ time algorithm using $\min(n, \log m)$ processors in the EREW-PRAM model. Section 4, which contains the heart of our CREW method, uses a new kind of sampling and pipelining, where samples are evenly spaced (and progressively finer) *clusters*

of elements and where the "help" for computing the information at a node does not come only from its children (as it did in [5]) but also from some of its subtree's leaves. Section 5 transforms the CREW algorithm of Section 4 into an EREW algorithm. This is not achieved by using known methods such as [6], but rather relies on (i) storing each leaf solution in a suitable parallel data structure, and (ii) re-defining the nature of the information stored at the internal nodes.

Recall that the EREW-PRAM is the parallel model where the processors operate synchronously and share a common memory, but no two of them are allowed simultaneous access to a memory cell (whether the access is for reading or for writing in that cell). The CREW-PRAM differs from the EREW-PRAM in that simultaneous reading is allowed (but simulateous writing is still forbidden).

2 Preliminaries

In this section we introduce some notation, terminology, and conventions.

Since the matrix A is understood, we henceforth use θ as a shorthand for θ_A. Throughout, R will denote the set of m row indices of A, and C will denote its n column indices. To avoid cluttering the exposition, we assume that m and n are powers of two (our scheme can easily be modified for the general case).

An *interval* of rows or columns is a non-empty set of *contiguous* (row or column) indices $[i,j] = \{i, i+1, \cdots, j\}$. We imagine row indices to lie on a horizontal line, so that a row is *to the left of* another row if and only if it has a smaller index (similarly "left of" is defined for columns). We say that interval I_1 is to the left of interval I_2, and I_2 is to the right of I_1, if the largest index of I_1 is smaller than the smallest index of I_2. (Note: for rows, it might seem more appropriate to say "above" rather than "to the left of", but the latter is in fact quite appropriate because we shall later map the rows to the leaves of a tree.)

Let I be a column interval, and let A_I be the $m \times |I|$ submatrix of A consisting of the columns of A in I. We use θ_I as a shorthand for θ_{A_I}. That is, if r is a row index, then $\theta_I(r)$ denotes the smallest $c \in I$ for which $A(r,c)$ is minimized. Note that $\theta_I(r)$ usually differs from $\theta(r)$, since we are minimizing only over I rather than C.

Throughout the paper, instead of storing θ_I directly, we shall instead store a function π_I which is an implicit description of θ_I.

Definition 2.1 *For any column interval I and any column index c, $\pi_I(c)$ is the row interval*

such that, for every row index r in that interval, we have $\theta_I(r) = c$; $\pi_I(c)$ is empty if no such r exists.

Note that the monotonicity of A implies that, if $c_1 < c_2$, then $\pi_I(c_1)$ is to the left of $\pi_I(c_2)$.

Note that each $\pi_I(c)$ can be stored in $O(1)$ space, since we need only store the beginning and end of that row interval. Throughout the paper, we shall use π_I as an implicit description of θ_I. The advantage of doing so is that we use $O(|I|)$ storage instead of the $O(|R|)$ that would be needed for explicitly storing θ_I. The disadvantage is that, given a row index r, a processor needs to binary search in the π_I array for the position of row index r in order to determine $\theta_I(r)$. Had we stored directly θ_I, $\theta_I(r)$ would be readily available in constant time. From now on, we consider our problem to be that of computing the π_C array. Once we have π_C, it is easy to do a postprocessing computation that obtains (explicitly) θ from π_C with m processors: each column c gets assigned $|\pi_C(c)|$ processors which set $\theta(r) = c$ for every $r \in \pi_C(c)$. Therefore Theorem 1.1 would easily follow if we can establish the following.

Theorem 2.1 *π_C can be computed in $O(\log m + \log n)$ time and $O(n)$ processors in the EREW-PRAM model.*

The rest of this paper proves the above theorem.

Definition 2.2 *The* s-sample *of the set R of row indices is obtained by choosing every s-th element of R (i.e., every row index which is a multiple of s). For example, the 4-sample of R is $(4,8,\cdots,m)$. For $k \in [0,\log m]$, let R_k denote the $(m/2^k)$-sample of R.*

For example, $R_0 = (m)$,
$R_3 = (m/8, m/4, 3m/8, m/2, 5m/8, 3m/4, 7m/8, m)$,
and $R_{\log m} = (1,2,\cdots,m) = R$.

Note that $|R_k| = 2^k = 2|R_{k-1}|$.

3 A $\log m$ Processor Algorithm

This section gives an algorithm which is needed as an important ingredient in the algorithm of the next section. It has the feature that its complexity bounds depend on the number of columns in a stronger way than on the number of rows.

Lemma 3.1 *The π_C array can be computed in $O(\max(n, \log m))$ time with $\min(n, \log m)$ processors in the EREW-PRAM model.*

The bounds of the above lemma might look unappealing at first sight, but their significance lies in the fact that m can be much larger than n. In fact we shall use this lemma, in the next section, on problems of size $m \times (\log n)$. The rest of this section proves the lemma. Simple-minded approaches like "use one processor to binary search for $\pi_C(c)$ in parallel for each $c \in C$" do not work, the difficulty being that we do not know which $\pi_C(c)$'s are empty. In fact, if we knew which $\pi_C(c)$'s are empty then we could easily achieve $O(\log m)$ time with n processors (by using the above-mentioned straightforward binary search—e.g., binary search for the right endpoint of $\pi_C(c)$ by doing $\log m$ comparisons involving the two columns c and c', where c' is the nearest column to the right of c having a nonempty $\pi_C(c')$).

We shall compute the π_C array by computing two arrays, $LeftExtend$ and $RightExtend$, whose significance is as follows.

Definition 3.1 *For any column c, let $LeftExtend(c)$ be the left endpoint of row interval $\pi_{[1,c]}(c)$. That is, $LeftExtend(c)$ is the minimum row index r such that, for any $c' < c$, $A(r,c) < A(r,c')$. Let $RightExtend(c)$ be the right endpoint of row interval $\pi_{[c,n]}(c)$. That is, $RightExtend(c)$ is the maximum row index r such that, for any $c' > c$, $A(r,c) \leq A(r,c')$.*

Intuitively, $LeftExtend(c)$ measures how far to the left (in R) column c can "extend its influence" if the only competition to it came from columns to its left. The intuition for $RightExtend(c)$ is analogous, with the roles of "left" and "right" being interchanged. Note that $LeftExtend(c)$ (resp., $RightExtend(c)$) might be undefined, which we denote by setting it equal to the nonexistent row $m+1$ (resp., 0).

Once we have the $RightExtend$ and $LeftExtend$ arrays, it is easy to obtain the π_C array, as follows. If either $LeftExtend(c) = m+1$ or $RightExtend(c) = 0$ then obviously $\pi_C(c)$ is empty. Otherwise we distinguish two cases: (i) if $RightExtend(c) < LeftExtend(c)$ then $\pi_C(c)$ is empty, and (ii) if $LeftExtend(c) \leq RightExtend(c)$ then interval $\pi_C(c)$ is not empty and has $LeftExtend(c)$ and $RightExtend(c)$ as its two endpoints. Hence it suffices to compute the $RightExtend$ and $LeftExtend$ arrays. The rest of this section explains how to compute the $LeftExtend$ array (the computation of $RightExtend$ is symmetrical and is therefore omitted). Furthermore, for the sake of definiteness, we shall describe our scheme assuming $n \geq \log m$ (it will be easy for the reader to see that it also works if $n < \log m$).

Thus we have $\min\{n, \log m\} = \log m$ processors and wish to achieve $O(\max\{n, \log m\}) = O(n)$ time performance. We shall show how to use the $\log m$ processors to compute the $LeftExtend$ array in $n + \log m$ $(= O(n))$ time steps. To simplify the presentation we assume, without loss of generality, that $A(m,1) > A(m,2) > \cdots > A(m,n)$ (one can always add a "dummy" last row to A in order to make this hold—obviously this does not destroy the monotonicity of A). This assumption simplifies the presentation because it causes every $LeftExtend(c)$ to be $\leq m$ (i.e., it is defined).

We first give a rough overview of our scheme. Imagine that the row indices R are organized as a complete binary search tree T_R: the leaves contain R sorted by increasing order, and each internal node v contains the row index r of the largest leaf in the subtree of v's left child (in which case we can simply refer to v as "internal node r" rather than the more cumbersome "the internal node that contains r"). Note that a row index $r < m$ appears exactly twice in T_R: once at a leaf, and once at an internal node (m appears only once, as the rightmost leaf). Having only $\log m$ processors, we clearly cannot afford to build all of T_R. Instead, we shall build a portion of it, starting from the root and expanding downwards along n root-to-leaf paths $P_1, \cdots, P_n$. Path P_c is in charge of computing $LeftExtend(c)$, and does so by performing a binary search for it as it traces a root-to-leaf path in the binary search tree T_R. If path P_c exits at leaf r then $LeftExtend(c) = r$. The tracing of all the P_i's is done in a total of $n + \log m$ time steps. Path P_c is inactive until time step c, at which time it gets assigned one processor and begins at the root, and at each subsequent step it makes one move down the tree, until it exits at some leaf at step $c + \log m$. Clearly there are at most $\log m$ paths that are simultaneously active, so that we use $\log m$ processors. At time step $n + \log m$ the last path (P_n) exits a leaf and the computation terminates. If, at a certain time step, path P_c wants to go down to a node of T_R not traced earlier by a $P_{c'}$ $(c' < c)$, then its processor builds that node of T_R (we use a pointer representation for the traced portion of T_R— we must avoid indexing since we cannot afford using m space). During path P_c's root-to-leaf trip, we shall maintain the property that, when P_c is at node r of T_R, $LeftExtend(c)$ is guaranteed to be one of the leaves in r's subtree (this property is obviously true when P_c is at the root, and we shall soon show how to maintain it as P_c goes from a node to one of its children). It is because of this property that the completion of a P_c (when it exits from a leaf of T_R) corresponds to the end of the computation of $LeftExtend(c)$.

The above overview implies that at each time step, the lowermost nodes of the $\log m$ active paths are "staggered" along the levels of T_R in that they are at levels $1, 2, \cdots, \log m$ respectively. Hence at each time step, at most one processor is active at each level of T_R, and the computation of exactly one $LeftExtend(c)$ gets completed (at one of the leaves).

We have omitted a crucial detail in the above overview: what additional information should be stored in the traced portion of T_R in order to aid the downward tracing of each P_c (such information is needed for P_c to know whether to branch left or right on its trip down). The idea is for each path to leave behind it a trail of information that will help subsequent paths. Note that $LeftExtend(c)$ depends upon columns $1, 2, \cdots, c-1$ and is independent of columns $c+1, \cdots, n$. Before specifying the additional information, we need some definitions.

Definition 3.2 *Let c and c' be column indices, r be a row index. We say that c is* better than c' for r *(denoted by $c <_r c'$) iff one of the following holds: (i) $A(r,c) < A(r,c')$, or (ii) $A(r,c) = A(r,c')$ and $c < c'$.*

Note that for any columns c, c' and row r, we must have either $c <_r c'$ or $c' <_r c$. When the algorithm compares $A(r,c)$ to $A(r,c')$ in order to determine whether $c <_r c'$ or $c' <_r c$, it is useful if the reader thinks of such a comparison as a *competition* between c and c' for r: c wins the competition over c' if the outcome is $c <_r c'$, otherwise it loses r to c'. When a path P_c is at an internal node r, it competes for r with the best column for r among the columns in $[1, c-1]$ (that is, it competes with $\theta_{[1,c-1]}(r)$). If c beats $\theta_{[1,c-1]}(r)$ in this competition for r, then P_c obviously branches down to the left child of r in T_R (its $LeftExtend(c)$ is certainly not greater than r). Otherwise it branches to the right child of r. However, the above assumes that the $\theta_{[1,c-1]}(r)$ values are available when needed. We must now make sure that, when P_c enters node r, it can easily (i.e., in constant time) obtain $\theta_{[1,c-1]}(r)$. Note that we can neither maintain the needed $\theta_{[1,c-1]}(r)$ at r, nor can we carry it down with P_c on its downward trip (it is not hard to see that either one of these two approaches runs into trouble). Instead, we shall use a judicious combination of both: some information is maintained locally in r, some is carried along by P_c. When P_c enters r, P_c combines the information it is carrying, with the information in r, to obtain in constant time $\theta_{[1,c-1]}(r)$. This is made more precise below.

Each internal node r' that has been already visited by a path contains, in a register $label(r')$, the best c' for r' among the subset of columns whose path went through r' (hence

label(r') is empty if no path visited r' so far).

When P_c enters internal node r of T_R, it carries with it, in a register *rival*(c), the largest c' such that $c' < c$ and $LeftExtend(c')$ is smaller than the leftmost leaf in the subtree of r (hence *rival*(c) is empty when P_c is at the root).

Note that the current *rival*(c) and *label*(r) allow P_c to obtain $\theta_{[1,c-1]}(r)$ in constant time, as follows. Recall that $\theta_{[1,c-1]}(r)$ is the best for r (i.e., smallest under the $<_r$ relationship) among the columns in $[1, c-1]$. Now, view $[1, c-1]$ as being partitioned into three subsets: the subset S_1 (resp., S_3) consisting of the columns c' whose $LeftExtend(c')$ is smaller (resp., larger) than the leftmost (resp., rightmost) leaf in the subtree of r, and the subset S_2 of the columns c' whose $LeftExtend(c')$ is in the subtree of r. Now, no column in S_3 can be $\theta_{[1,c-1]}(r)$ (by the definition of S_3). The best for r in S_2 is *label*(r) (by definition). The best for r in S_1 is *rival*(c), by the following argument. Suppose to the contrary that there is a $c'' \in S_1$, $c'' <$ *rival*(c), such that c'' is better for r than *rival*(c). A contradiction with the monotonicity of A is obtained by observing that we now have: (i) $c'' <$ *rival*(c), (ii) $LeftExtend($*rival*$(c)) < r$, and (iii) c'' is better than *rival*(c) for r but not for $LeftExtend($*rival*$(c))$. Hence *rival*(c) must be the best for r in S_1. Hence $\theta_{[1,c-1]}(r)$ is one of {*rival*(c),*label*(r)}, which can be obtained in constant time from *rival*(c) and *label*(r), both of which are available (P_c carried *rival*(c) down with it when it entered r, and r itself maintained *label*(r) during the previous time step).

The main problem that remains is how to update, in constant time, the *rival*(c) and the *label*(r) registers when P_c goes from r down to one of r's two children. We explain below how P_c updates its *rival*(c) register, and how r updates its *label*(r) register. (We need not worry about updating the *label*(r') of a row r' not currently being visited by a $P_{c'}$, since such a *label*(r') remains by definition unchanged.)

By its very definition, *label*(r) depends on all the paths $P_{c'}$ ($c' < c$) that previously went through r. Since P_c has just visited r, we need to make c compete, for row r, with the previous value of *label*(r): if c wins then *label*(r) becomes c, otherwise it remains unchanged.

The updating of *rival*(c) depends upon one of the following two cases.

The first case is when c won the competition at r, i.e., P_c has moved to the left child of r (call it r'). In that case by its very definition *rival*(c) remains unchanged (since the leftmost leaf in the subtree of r' is the same as the leftmost leaf in the subtree of r).

The second case is when c lost the competition at r, i.e., P_c has moved to the right child of r (call it r''). In that case we claim that it suffices to compare the old *label*(r) to the

old $rival(c)$: the one which is better for r'' is the new value of $rival(c)$. We now prove the claim. Let r_1 (resp., r_2) be the leftmost leaf in the subtree of r (resp., r''). Let C' (resp., C'') be the set of paths consisting of the columns β such that $\beta < c$ and $LeftExtend(\beta) < r_1$ (resp., $LeftExtend(\beta) < r_2$). By definition, the old (resp., new) value of $rival(c)$ is the largest column index in C' (resp., C''). The claim would follow if we can prove that the largest column index in $C'' - C'$ is the old $label(r)$ (by "old $label(r)$" we mean its value before updating, i.e., its value when P_c first entered r). We prove this by contradiction: let $\hat{c}$ denote the old $label(r)$, and suppose that there is a $\gamma \in C'' - C'$ such that $\gamma > \hat{c}$. Since both γ and $\hat{c}$ are in $C'' - C'$, their respective paths went from r to the left child of r. However, P_γ did so *later* than $P_{\hat{c}}$ (because $\gamma > \hat{c}$). This in turn implies that γ is better for r than $\hat{c}$, contradicting the fact that $\hat{c}$ is the old $label(r)$. This proves the claim.

Concerning the implementation of the above scheme, the assignment of processors to their tasks is trivial: each active P_c carries with it its own processor, and when it exits from a leaf it releases that processor which gets assigned to $P_{c+\log m}$ which is just beginning at the root of T_R.

4 The Tree-Based Algorithm

This section builds on the algorithm of the previous section and establishes the CREW version of Theorem 2.1 (the next section will extend it to EREW). The sampling and iterative refinement methods used in this section are reminiscent of the cascading divide-and-conquer technique [5, 4], and are similar to those used in [3] for solving in parallel the string editing problem. In fact, although [3] deals with a different problem, it implicitly contains a solution to our problem whose complexity is also $O(\log n)$ time but with a disappointing $O(n \log n)$ processors. As in [3], it is useful to think of the computation as progressing through the nodes of a tree T which we now proceed to define (it differs substantially from [3]).

Partition the column indices into $n/\log n$ adjacent intervals $I_1, \cdots, I_{n/\log n}$ of size $\log n$ each. Call each such interval I_i a *fat column*. Imagine a complete binary tree T on top of these fat columns, and associate with each node v of this tree a *fat interval* $I(v)$ (i.e., an interval of fat columns) in the following way: the fat interval associated with a leaf is simply the fat column corresponding to it, and the fat interval associated with an internal node is the union of the two fat intervals of its children. Thus a node v at height h has a fat

interval $I(v)$ consisting of $|I(v)| = 2^h$ fat columns. The storage representation we use for a fat interval $I(v)$ is a list containing the indices of the fat columns in it; we also call that list $I(v)$, in order to avoid introducing extra notation. For example, if v is the left child of the root, then the $I(v)$ array contains $(1, 2, \cdots, n/(2\log n))$. Observe that $\sum_{v \in T} |I(v)| = O(|T| \log |T|) = O((n/\log n) \log n) = O(n)$.

The ultimate goal is to compute $\pi_C(c)$ for every $c \in C$.

Let *leaf problem* I_i be the problem of computing $\pi_{I_i}(c)$ for all $c \in I_i$. Thus a leaf problem is a subproblem of size $m \times \log n$. From Lemma 3.1 it follows that a leaf problem can be solved in $O(\log n + \log m)$ time by $\min\{\log n, \log m\}$ ($\leq \log n$) processors. Since there are $n/\log n$ leaf problems, they can be solved in $O(\log n + \log m)$ time by n processors. We assume that this has already been done, i.e., that we know the π_{I_i} array for each leaf problem I_i. The rest of this section shows that an additional $O(\log n + \log m)$ time with $O(n)$ processors is enough for obtaining π_C.

Definition 4.1 *Let $J(v)$ be the interval of original columns that belong to fat intervals in $I(v)$ (hence $|J(v)| = |I(v)| \cdot \log n$). For every $v \in T$, fat column $f \in I(v)$, and subset R' of R, let $\psi_v(R', f)$ be the interval in R' such that, for every r in that interval, $\theta_{J(v)}(r)$ is a column in fat column f. We use "$\psi_v(R', *)$" as a shorthand for "$\psi_v(R', f)$ for all $f \in I(v)$)".*

We henceforth focus on the computation of the $\psi_{root(T)}(R, *)$ array, where $root(T)$ is the root node of T. Once we have the array $\psi_{root(T)}(R, *)$, it is easy to compute the required π_C array within the prescribed complexity bounds: for each fat column f, we replace the $\psi_{root(T)}(R, f)$ row interval by its intersection with the row intervals in the π_{I_f} array (which are already available at the leaf I_f of T). The rest of this section proves the following lemma.

Lemma 4.1 *$\psi_v(R, *)$ for every $v \in T$ can be computed by a CREW-PRAM in time $O(height(T) + \log m)$ and $O(n)$ processors, where $height(T)$ is the height of T ($= O(\log n)$).*

Proof. Since $\sum_{v \in T}(|I(v)| + \log n) = O(n)$, we have enough processors to assign $|I(v)| + \log n$ of them to each $v \in T$. The computation proceeds in $\log m + height(T) - 1$ *stages*, each of which takes constant time. Each $v \in T$ will compute $\psi_v(R', *)$ for progressively larger subsets R' of R, subsets R' that double in size from one stage to the next of the computation.

We now state precisely what these subsets are. Recall that R_i denotes the $(m/2^i)$-sample of R, so that $|R_i| = 2^i$.

At the t-th stage of the algorithm, a node v of height h in T will use its $|I(v)| + \log n$ processors to compute, in constant time, $\psi_v(R_{t-h}, *)$ if $h \leq t \leq h + \log m$. It does so with the help of information from $\psi_v(R_{t-1-h}, *)$, $\psi_{LeftChild(v)}(R_{t-h}, *)$, and $\psi_{RightChild(v)}(R_{t-h}, *)$, all of which are available from the previous stage $t-1$ (note that $(t-1)-(h-1) = t-h$). If $t < h$ or $t > h + \log m$ then node v does nothing during stage t. Thus before stage h the node v lies "dormant", then at stage $t = h$ it first "wakes up" and computes $\psi_v(R_0, *)$, then at the next stage $t = h+1$ it computes $\psi_v(R_1, *)$, etc. At stage $t = h + \log m$ it computes $\psi_v(R_{\log m}, *)$, after which it is done.

The details of what information v stores and how it uses its $|I(v)| + \log n$ processors to perform stage t in constant time are given below. In the description, tree nodes u and w are the left and right child, respectively, of v in T.

After stage t, node v (of height h) contains $\psi_v(R_{t-h}, *)$ and a quantity $Critical_v(R_{t-h})$ whose significance is as follows.

Definition 4.2 *Let R' be any subset of R. $Critical_v(R')$ is the largest $r \in R'$ that is contained in $\psi_v(R', f)$ for some $f \in I(u)$; if there is no such r then $Critical_v(R') = 0$.*

The monotonicity of A implies that for every $r' < Critical_v(R')$ (resp., $r' > Critical_v(R')$), r' is contained in $\psi_v(R', f)$ for some $f \in I(u)$ (resp., $f \in I(w)$).

We now explain how v performs stage t, i.e., how it obtains $Critical_v(R_{t-h})$ and $\psi_v(R_{t-h}, *)$ using $\psi_u(R_{t-h}, *)$, $\psi_w(R_{t-h}, *)$, and $Critical_v(R_{t-1-h})$ (all three of which were computed in the previous stage $t-1$). The fact that the $|I(v)| + \log n$ processors can do this in constant time is based on the following observations, whose correctness follows from the definitions.

Observation 4.1 *1. $Critical_v(R_{t-h})$ is either the same as $Critical_v(R_{t-1-h})$, or the successor of $Critical_v(R_{t-1-h})$ in R_{t-h}.*

2. If $f \in I(u)$, then $\psi_v(R_{t-h}, f)$ is the portion of interval $\psi_u(R_{t-h}, f)$ that is $\leq Critical_v(R_{t-h})$.

3. If $f \in I(w)$, then $\psi_v(R_{t-h}, f)$ is the portion of interval $\psi_w(R_{t-h}, f)$ that is $> Critical_v(R_{t-h})$.

The algorithmic implications of the above observations are discussed next.

4.1 Computing $Critical_v(R_{t-h})$.

Relationship (1) of Observation 4.1 implies that, in order to compute $Critical_v(R_{t-h})$, all v has to do is determine which of $Critical_v(R_{t-1-h})$ or its successor in R_{t-h} is the correct value of $Critical_v(R_{t-h})$. This is done as follows. If $Critical_v(R_{t-1-h})$ has no successor in R_{t-h} then $Critical_v(R_{t-1-h}) = m$ (the last row) and hence $Critical_v(R_{t-h}) = Critical_v(R_{t-1-h})$. Otherwise the updating is done in the following two steps. For conciseness, let r denote $Critical_v(R_{t-1-h})$, and let s denote the successor of r in R_{t-h}.

- The first step is to compute $\theta_{J(u)}(s)$ and $\theta_{J(w)}(s)$ in constant time. This involves a search in $I(u)$ (resp., $I(w)$) for the fat column $f' \in I(u)$ (resp., $f'' \in I(w)$) whose $\psi_u(R_{t-h}, f')$ (resp., $\psi_w(R_{t-h}, f'')$) contains s. These two searches in $I(u)$ and $I(w)$ are done in constant time with the $|I(v)|$ processors available. We explain how the search for f' in $I(u)$ is done (that for f'' in $I(w)$ is similar and omitted). Node v assigns a processor to each $f \in I(u)$, and that processor tests whether s is in $\psi_u(R_{t-h}, f)$; the answer is "yes" for exactly one of those $|I(u)|$ processors and thus can be collected in constant time. Next, v determines $\theta_{J(u)}(s)$ and $\theta_{J(w)}(s)$ in constant time by using $\log n$ processors to search for s in constant time in the leaf solutions $\pi_{I_{f'}}$ and $\pi_{I_{f''}}$ available at leaves f' and f'', respectively. If the outcome of the search for s in $\pi_{I_{f'}}$ is that $s \in \pi_{I_{f'}}(c')$ for $c' \in I_{f'}$, then $\theta_{J(u)}(s) = c'$. Similarly, $\theta_{J(w)}(s)$ is obtained from the outcome of the search for s in $\pi_{I_{f''}}$.

- The next step consists of comparing $A(s, \theta_{J(u)}(s))$ to $A(s, \theta_{J(w)}(s))$. If the outcome is $A(s, \theta_{J(u)}(s)) > A(s, \theta_{J(w)}(s))$, then $Critical_v(R_{t-h})$ is the same as $Critical_v(R_{t-1-h})$. Otherwise $Critical_v(R_{t-h})$ is s.

We next show how the just computed $Critical_v(R_{t-h})$ value is used to compute $\psi_v(R_{t-h}, *)$ in constant time.

4.2 Computing $\psi_v(R_{t-h}, *)$.

Relationship (2) of Observation 4.1 implies the following for each $f \in I(u)$:

- If $\psi_u(R_{t-h}, f)$ is to the left of $Critical_v(R_{t-h})$, then $\psi_v(R_{t-h}, f) = \psi_u(R_{t-h}, f)$.

- If $\psi_u(R_{t-h}, f)$ is to the right of $Critical_v(R_{t-h})$ then $\psi_v(R_{t-h}, f) = \emptyset$.

- If $\psi_u(R_{t-h}, f)$ contains $Critical_v(R_{t-h})$ then $\psi_v(R_{t-h}, f)$ consists of the portion of $\psi_u(R_{t-h}, f)$ up to (and including) $Critical_v(R_{t-h})$.

The above three facts immediately imply that $O(1)$ time is enough for $|I(u)|$ of the $|I(v)|$ processors assigned to v to compute $\psi_v(R_{t-h}, f)$ for all $f \in I(u)$ (recall that the $\psi_u(R_{t-h}, *)$ array is available in u from the previous stage $t-1$, and $Critical_v(R_{t-h})$ has already been computed).

A similar argument, using relationship (3) of Obervation 4.1, shows that $|I(w)|$ processors are enough for computing $\psi_v(R_{t-h}, f)$ for all $f \in I(w)$. Thus $\psi_v(R_{t-h}, *)$ can be computed in constant time with $|I(v)|$ processors.

This completes the proof of Lemma 4.1. □

5 Avoiding Read Conflicts

The scheme of the previous section made crucial use of the "concurrent read" capability of the CREW-PRAM. This occurred in the computation of $Critical_v(R_{t-h})$ and also in the subsequent computation of $\psi_v(R_{t-h}, *)$. In its computation of $Critical_v(R_{t-h})$, there are two places where the algorithm of the previous section uses the "concurrent read" capability of the CREW (both of them occur during the computation of $\theta_{J(u)}(s)$ and $\theta_{J(w)}(s)$ in constant time). After that, the CREW part of the computation of $\psi_v(R_{t-h}, *)$ is the common reading of $Critical_v(R_{t-h})$. We review these three problems next, using the same notation as in the previous section (i.e., u is the left child of v in T, w is the right child of v in T, v has height h in T, s is the successor of $Critical_v(R_{t-1-h})$ in R_{t-h}, etc.).

- Problem 1: This arises during the search in $I(u)$ (resp., $I(w)$) for the fat column $f' \in I(u)$ (resp., $f'' \in I(w)$) whose $\psi_u(R_{t-h}, f')$ (resp., $\psi_w(R_{t-h}, f'')$) contains s. Specifically, for finding (e.g.) f', node v assigns a processor to each $f \in I(u)$, and that processor tests whether s is in $\psi_u(R_{t-h}, f)$; the answer is "yes" for exactly one of those $|I(u)|$ processors and thus can be collected in constant time.

- Problem 2: Having found f' and f'', node v determines $\theta_{J(u)}(s)$ and $\theta_{J(w)}(s)$ in constant time by using $\log n$ processors to search for s, in constant time, in the leaf solutions $\pi_{I_{f'}}$ and $\pi_{I_{f''}}$ available at leaves f' and f'', respectively. There are two parts to this problem: (i) many ancestors of a leaf I_f (possibly all $\log n$ of them) may simultaneously access the same leaf solution π_{I_f}, and (ii) each of those ancestors uses

$\log n$ processors to do a constant-time search in the leaf solution π_{I_f} (in the EREW model, it would take $\Omega(\log\log n)$ time just to tell the processors in which leaf to search).

- Problem 3: During the computation of $\psi_v(R_{t-h},*)$, the common reading of the $Critical_v(R_{t-h})$ value by the fat columns $f \in I(v)$.

Any solution we design for Problems 1–3 should also be such that no concurrent reading of an entry of matrix A occurs. We begin by discussing how to handle Problem 2.

5.1 Problem 2.

To avoid the "many ancestors" part of Problem 2 (i.e., part (i)), it naturally comes to mind to make $\log n$ copies of each leaf I_f and to dedicate each copy to one ancestor of I_f, especially since we can easily create these $\log n$ copies of I_f in $O(\log n)$ time and $\log n$ processors (because the space taken by π_{I_f} is $O(\log n)$). But we are still left with part (ii) of Problem 2, i.e., how an ancestor can search the copy of I_f dedicated to it in constant time by using its $\log n$ processors in an EREW fashion. On the one hand, just telling all of those $\log n$ processors which I_f to search takes an unacceptable $\Omega(\log\log n)$ time, and on the other hand a single processor seems unable to search π_{I_f} in constant time. We resolve this by organizing the information at (each copy of) I_f in such a way that we can replace the $\log n$ processors by a single processor to do the search in constant time. Instead of storing a leaf solution in an array π_{I_f}, we store it in a tree structure (call it $Tree(f)$) that enables us to exploit the highly structured nature of the searches to be performed on it. The search to be done at any stage t is not arbitrary, and is highly dependent on what happened during the previous stage $t-1$, which is why a single processor can do it in constant time (as we shall soon see).

We now define the tree $Tree(f)$. Let $List = [1,r_1],[r_1+1,r_2],\cdots,[r_p+1,m]$ be the list of (at most $\log n$) nonempty intervals in π_{I_f}, in sorted order. Each node of $Tree(f)$ contains one of the intervals of $List$. (It is implicitly assumed that the node of $Tree(f)$ that contains $\pi_{I_f}(c)$ also stores its associated column c.) Imagine a procedure that builds $Tree(f)$ from the root down, in the following way (this is not how $Tree(f)$ is actually built, but it is a convenient way of defining it). At a typical node x, the procedure has available a contiguous subset of $List$ (call it $L(x)$), together with an integer $d(x)$, such that no interval of $L(x)$ contains a multiple of $m/(2^{d(x)})$. (The procedure starts at the root of $Tree(f)$ with

Figure 1. Illustrating $Tree(f)$.

$L(root) = List$ and $d(root) = -1$.) The procedure determines which interval of $L(x)$ to store in x by finding the smallest integer $i > d(x)$ such that a multiple of $m/(2^i)$ is in an interval of $L(x)$ (we call i the *priority* of that interval), together with the interval of $L(x)$ for which this happens (say it is interval $[r_k + 1, r_{k+1}]$, and note that it is unique). Interval $[r_k + 1, r_{k+1}]$ is then stored at x (together with its associated column), and the subtree of x is created recursively, as follows. Let L' (resp., L'') be the portion of $L(x)$ to the left (resp., right) of interval $[r_k + 1, r_{k+1}]$. If $L' \neq \emptyset$ then the procedure creates a left child for x (call it y) and recursively goes to y with $d(y) = i$ and with $L(y) = L'$. If $L'' \neq \emptyset$ then the procedure creates a right child for x (call it z) and recursively goes to z with $d(z) = i$ and with $L(z) = L''$.

Note that the root of $Tree(f)$ has priority zero and no right child, and that its left child w has $d(w) = 0$ and $L(w) =$ (*List* minus the last interval in *List*).

Figure 1 shows the $Tree(f)$ corresponding to the case where $m = 32$, $\log n = 8$, and

$$\pi_{I_f} = [1,6]\ [7,8]\ [9,14]\ [\,]\ [15,15]\ [16,17]\ [18,22]\ [23,32].$$

In that figure, we have assumed (for convenience) that the columns in I_f are numbered $1, \cdots, 8$, and we have shown both the columns c (circled) and their associated intervals $\pi_{I_f}(c)$. For this example, the priorities of the nonempty intervals of π_{I_f} are (respectively) $3, 2, 3, 5, 1, 3, 0$. The concept of priority will be useful for building $Tree(f)$ and for proving various facts about it. The following proposition is an easy consequence of the above definition of $Tree(f)$.

Proposition 5.1 *Let X be an interval in List, of priority i. Let X' (resp., X'') be the nearest interval that is to the left (resp., right) of X in List and that has priority smaller*

than i. Let i' (resp., i'') be the priority of X' (resp., X''). Then we have the following:

1. *If $i > 0$ then at least one of $\{X', X''\}$ exists.*
2. *If only one of $\{X', X''\}$ exists, then X is its child in $Tree(f)$ (right child in case of X', left child in case of X'').*
3. *If X' and X'' both exist, then $i' \neq i''$. Furthermore, if $i' > i''$ then X is the right child of X' in $Tree(f)$, otherwise (i.e., if $i' < i''$) X is the left child of X''.*

Proof. The proof refers to the hypothetical procedure we used to define $Tree(f)$. For convenience, we denote a node x of $Tree(f)$ by the interval X that it contains; i.e., whereas in the description of the procedure we used to say "the node x of $Tree(f)$ that contains X" and "list $L(x)$", we now simply say "node X" and "list $L(X)$" (this is somewhat of an abuse of notation, since when the procedure first entered x it did not yet know X).

That (1) holds is obvious.

That (2) holds follows from the following observation. Let X be a child of X' in $Tree(f)$. When the procedure we used to define $Tree(f)$ was at X', it went to node X with the list $L(X)$ set equal to the portion of $L(X')$ before X' (if X is left child of X') or after X' (if X is right child of X'). This implies that the priorities of the intervals between X' and X in *List* are all larger than the priority of X. This implies that, when starting in *List* at X and moving along *List* towards X', X' is the first interval that we encounter that has a lower priority than that of X. Hence (2) holds.

We now prove (3). Note that the proof we just gave for (2) also implies that the parent of X is in $\{X', X''\}$. Hence to prove (3), it suffices to show that one of $\{X', X''\}$ is ancestor of the other (this would imply that they are both ancestors of X, and that the one with the larger priority is the parent of X). We prove this by contradiction: let Z be the lowest common ancestor of X' and X'' in $Tree(f)$, with $Z \notin \{X', X''\}$. Since Z has lower priority than both X' and X'', it cannot occur between X' and X'' in *List*. However, the fact that X' (resp., X'') is in the subtree of the left (resp., right) child of Z implies that it is in the portion of $L(Z)$ before Z (resp., after Z). This implies that Z occurs between X' and X'' in *List*, a contradiction. □

We now show that $Tree(f)$ can be built in $O(\log m + \log n)$ time with $|List|$ ($\leq \log n$) processors. Assign one processor to each interval of *List*, and do the following in parallel for each such interval. The processor assigned to an interval X computes the smallest

integer i such that a multiple of $m/(2^i)$ is in that interval (recall that this integer i is the priority of interval X). Following this $O(\log m)$ time computation of the priorities of the intervals in $List$, the processor assigned to each interval determines its parent in $Tree(f)$, and whether it is left or right child of that parent, by using the above Proposition 5.1. Reading conflicts are easy to avoid during this $O(|List|)$ time computation (the details are trivial and omitted).

Note: Although we do not need to do so, it is in fact possible to build $Tree(f)$ in $O(\log m + \log n)$ time by using only one processor rather than $\log n$ processors, but the construction is somewhat more involved and we refrain from giving it in order not to break the flow of the exposition.

As explained earlier, after $Tree(f)$ is built, we must make $\log n$ copies of it (one for each ancestor in T of leaf I_f).

We now explain how a single processor can use a copy of $Tree(f)$ to perform constant time searching. From now on, if a node of $Tree(f)$ contains interval $\pi_{I_f}(c)$, we refer to that node as either "node $\pi_{I_f}(c)$" or as "node c". We use $RightChild(c)$ and $LeftChild(c)$ to denote the left and right child, respectively, of c in $Tree(f)$.

Proposition 5.2 *Let $r \in R_k$, r' be the predecessor of r in R_k. Let $r \in \pi_{I_f}(c)$, and let $r' \in \pi_{I_f}(c')$. Then the predecessor of r in R_{k+1} (call it r'') is in $\pi_{I_f}(c'')$ where $c'' \in \{c', RightChild(c'), LeftChild(c), c\}$.*

Proof. If $c'' \in \{c', c\}$ then there is nothing to prove, so suppose that $c'' \notin \{c, c'\}$. This implies that $c \neq c'$ (since $c = c'$ would imply that $c'' = c$). Then $\pi_{I_f}(c')$, $\pi_{I_f}(c'')$ and $\pi_{I_f}(c)$ are distinct and occur in that order in $List$ (not necessarily adjacent to one another in $List$). Note that $\pi_{I_f}(c')$ and $\pi_{I_f}(c)$ each contains a row in R_k, that $\pi_{I_f}(c'')$ contains a row in R_{k+1} but no row in R_k, and that all the other intervals of $List$ that are between $\pi_{I_f}(c')$ and $\pi_{I_f}(c)$ do not contain any row in R_{k+1}. This, together with the definition of the priority of an interval, implies that $\pi_{I_f(c')}$ and $\pi_{I_f(c)}$ have priorities no larger than k, that $\pi_{I_f}(c'')$ has a priority equal to $k+1$, and that all the other intervals of $List$ that are between $\pi_{I_f}(c')$ and $\pi_{I_f}(c)$ have priorities greater than $k+1$. This, together with proposition 5.1, implies that $c'' \in \{RightChild(c'), LeftChild(c)\}$. □

Proposition 5.3 *Let $r \in R_k$, r' be the successor of r in R_k. Let $r \in \pi_{I_f}(c)$, and let $r' \in \pi_{I_f}(c')$. Then the successor of r in R_{k+1} (call it r'') is in $\pi_{I_f}(c'')$ where $c'' \in \{c, RightChild(c), LeftChild(c'), c'\}$.*

Proof. Similar to that of Proposition 5.2, and therefore omitted. □

Now recall that, in the previous section, the searches done by a particular v in a leaf solution at I_f had the feature that, if at stage t node $v \in T$ asked which column of I_f contains a certain row $r \in R_k$, then at stage $t+1$ it is asking the same question about r' where r' is either the successor or the predecessor of r in R_{k+1}. This, together with Propositions 5.2 and 5.3, implies that a processor can do the search (in $Tree(f)$) at stage $t+1$ in constant time, so long as it maintains, in addition to the node of $Tree(f)$ that contains the current r, the two nodes of $Tree(f)$ that contain the predecessor and (respectively) successor of r in R_k. These are clearly easy to maintain. □

We next explain how Problems 1 and 3 are handled.

5.2 Problems 1 and 3.

Right after stage $t-1$ is completed, v stores the following information (recall that the height of v in T is h). The fat columns $f \in I(v)$ for which interval $\psi_v(R_{t-1-h}, f)$ is not empty are stored in a doubly linked list. For each such fat column f, we store the following information: (i) the row interval $\psi_v(R_{t-1-h}, f) = [\alpha_1, \alpha_2]$, and (ii) for row α_1 (resp., α_2), a pointer to the node, in v's copy of $Tree(f)$, whose interval contains row α_1 (resp., α_2). In addition, v stores the following. Let z be the parent of v in T, and let s' be the successor of $Critical_z(R_{t-1-(h+1)})$ in $R_{t-(h+1)}$, with s' in $\psi_v(R_{t-1-h}, g)$ for some $g \in I(v)$. Then v has g marked as being its *distinguished fat column*, and v also stores a pointer to the node, in v's copy of $Tree(g)$, whose interval contains s'. Of course, information similar to the above for v is stored in every node x of T (including v's children, u and w), with the height of x playing the role of h. In particular, in the formulation of Problem 1, the fat columns we called f' and f'' are the distinguished fat columns of u and (respectively) w, and thus are available at u and (respectively) w, each of which also stores a pointer to the node containing s in its copy of $Tree(f')$ (for u) or of $Tree(f'')$ (for w).

Assume for the time being that we are able to maintain the above information in constant time from stage $t-1$ to stage t. This would enable us to avoid Problem 1 because instead of searching for the desired fat column f' (resp., f''), node u (resp., w) already has it available as its distinguished fat column. Problem 3 would also be avoided, because now only the distinguished fat columns of u and w need to read from v the $Critical_v(R_{t-h})$ value (whereas previously all of the fat columns in $I(u) \cup I(w)$ read that value from v). It therefore suffices to show how to maintain, from stage $t-1$ to stage t, the above information (i.e., v's linked

list and its associated pointers to the $Tree(f)$s of its elements, v's distinguished fat column g, and the pointer to v's copy of $Tree(g)$). We explain how this is done at v.

First, v computes its $Critical_v(R_{t-h})$: since we know from stage $t-1$ the distinguished fat columns f' and f'' of u and (respectively) w, and their associated pointers to u's copy of $Tree(f')$ and (respectively) w's copy of $Tree(f'')$, v can compare the two relevant entries of matrix A (i.e., $A(s,\theta_{J(u)}(s))$ and $A(s,\theta_{J(w)}(s))$) and it can decide, based on this comparison, whether $Critical_v(R_{t-h})$ remains equal to $Critical_v(R_{t-1-h})$ or becomes equal to s, its successor in R_{t-h}. But since this is done in parallel by all v's, we must show that no two nodes of T (say, v and v') try to access the same entry of matrix A. The reason this does not happen is as follows. If none of $\{v,v'\}$ is ancestor of the other then no read conflict in A can occur between v and v' because their associated columns (that is, $J(v)$ and $J(v')$) are disjoint. If one of v,v' is ancestor of the other, then no read conflict in A can occur between them because the rows they are interested in are disjoint (this is based on the observation that v is interested in rows in $R_{t-h} - \cup_{i=0}^{t-1-h} R_i$ and hence will have no conflict with any of its ancestors).

As a side effect of the computation of $Critical_v(R_{t-h})$, v also knows which fat column $\hat{f} \in \{f', f''\}$ is such that $\psi_v(R_{t-h}, \hat{f})$ contains $Critical_v(R_{t-h})$. It uses its knowledge of $\hat{f}$ to update its linked list of nonempty fat columns (and their associated row intervals) as follows (we distinguish two cases):

1. $\hat{f} = f'$. In that case v's new linked list consists of the portion of u's old (i.e., at stage $t-1$) linked list whose fat columns are $< f'$ (the row intervals associated with these fat columns are as in u's list at $t-1$), followed by f' with an associated interval $\psi_v(R_{t-h}, f')$ equal to the portion of $\psi_u(R_{t-h}, f')$ up to and including $Critical_v(R_{t-h})$, followed by f'' with an associated interval $\psi_v(R_{t-h}, f'')$ equal to the portion of $\psi_w(R_{t-h}, f'')$ larger than $Critical_v(R_{t-h})$ if that portion is nonempty (if it is empty then f'' is not included in v's new linked list), followed by the portion of w's old (i.e., at stage $t-1$) linked list whose fat columns are $> f''$ (the row intervals associated with these fat columns are as in w's list at $t-1$).

2. $\hat{f} = f''$. Similar to the first case, except that $Critical_v(R_{t-h})$ is now in the interval associated with f'' rather than f'.

It should be clear that the above computation of v's new linked list and its associated intervals can be implemented in constant time with $|I(v)|$ processors (by copying the needed

information from u and w, since these are at height $h-1$ and hence at stage $t-1$ already "knew" their information relative to $R_{t-1-(h-1)} = R_{t-h}$).

In either one of the above two cases (1) and (2), for each endpoint α of a $\psi_v(R_{t-h}, f)$ in v's linked list, we must compute the pointer to the node, in v's copy of $Tree(f)$, whose interval contains that α. We do it as follows. If f was not in v's list at stage $t-1$, then we obtain the pointer from u or w, simply by copying it (more specifically, if in u or w it points to a node of u's or w's copy of $Tree(f)$, then the "copy" we make of that pointer is to the same node but in v's own copy of $Tree(f)$). On the other hand, if f was in v's list at stage $t-1$, then we distinguish two cases. In the case where α was also an endpoint of $\psi_v(R_{t-1-h}, f)$, we already have its pointer (to v's copy of $Tree(f)$) available from stage $t-1$. If α was not an endpoint of $\psi_v(R_{t-1-h}, f)$ then α is predecessor or successor of an endpoint of $\psi_v(R_{t-1-h}, f)$ in R_{t-h}, and therefore the pointer for α can be found in constant time (by using Propositions 5.1 and 5.2).

Finally, we must show how v computes its new distinguished fat column. It does so by first obtaining, from its parent z, $Critical_z(R_{t-(h+1)})$ that z has just computed. The old distinguished fat column g stored at v had its $\psi_v(R_{t-1-h}, g)$ containing the successor s' of $Critical_z(R_{t-1-(h+1)})$ in $R_{t-(h+1)}$. It must be updated into a $\hat{g}$ such that $Critical_v(R_{t-h}, \hat{g})$ contains the successor s'' of $Critical_z(R_{t-(h+1)})$ in $R_{t+1-(h+1)}$. We distinguish two cases.

1. $Critical_z(R_{t-(h+1)}) = Critical_z(R_{t-1-(h+1)})$. In that case s'' is the predecessor of s' in $R_{t+1-(h+1)}$, and the fat column $\hat{g} \in I(v)$ for which $\psi_v(R_{t-h}, \hat{g})$ contains s'' is either the same as g or it is the predecessor of g in the linked list for v at stage t (which we already computed). It is therefore easy to identify $\hat{g}$ in constant time in that case.

2. $Critical_z(R_{t-(h+1)}) = s'$. In that case s'' is the successor of s' in $R_{t+1-(h+1)}$, and the fat column $\hat{g} \in I(v)$ for which $\psi_v(R_{t-h}, \hat{g})$ contains s'' is either the same as g or it is the successor of g in the linked list for v at stage t (which we already computed). It is therefore easy to identify $\hat{g}$ in constant time in that case as well.

In either one of the above two cases, we need to also compute a pointer value to the node, in v's copy of $Tree(\hat{g})$, whose interval contains s''. This is easy if $\hat{g} = g$, because we know from stage $t-1$ the pointer value for s' into v's copy of $Tree(g)$, and the pointer value for s'' can thus be found by using Propositions 5.2 and 5.3 (since s'' is predecessor or successor of s' in R_{t-h}). So suppose $\hat{g} \neq g$, i.e., $\hat{g}$ is predecessor or successor of g in v's

linked list of nonempty fat columns at t. The knowledge of the pointer for s' to v's copy of $Tree(g)$ at $t-1$ is of little help in that case, since we now care about v's copy of $Tree(\hat{g})$ rather than $Tree(g)$. What saves us is the following observation: *s'' must be an endpoint* of row interval $\psi_v(R_{t-h}, \hat{g})$. Specifically, s'' is the left (i.e., beginning) endpoint of $\psi_v(R_{t-h}, \hat{g})$ if $\hat{g}$ is the successor of g in v's linked list at stage t (Case 2 above), otherwise it is the right endpoint of $\psi_v(R_{t-h}, \hat{g})$ (Case 1 above). Since s'' is such an endpoint, we already know the pointer for s'' to v's copy of $Tree(\hat{g})$ (because such pointers are available, in v's linked list, for all the endpoints of the row intervals in that linked list).

References

[1] A. Aggarwal and J. Park, "Parallel searching in multidimensional monotone arrays," to appear in *J. of Algorithms.* (A preliminary version appeared in *Proc. 29th Annual IEEE Symposium on Foundations of Computer Science*, 1988, pp. 497–512.)

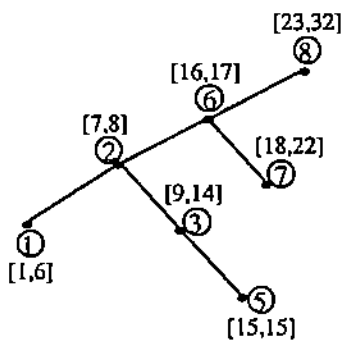
[2] A. Aggarwal, M. M. Klawe, S. Moran, P. Shor and R. Wilber, "Geometric Applications of a Matrix Searching Algorithm," *Algorithmica*, Vol. 2, pp. 209–233, 1987.

[3] A. Apostolico, M. J. Atallah, L. Larmore, and H. S. McFaddin, "Efficient Parallel Algorithms for String Editing and Related Problems," *SIAM J. Comput.*, Vol. 19, pp. 968-988, 1990.

[4] M.J. Atallah, R. Cole and M.T. Goodrich, "Cascading Divide-and-Conquer: A Technique for Designing Parallel Algorithms," *SIAM J. Comput.*, Vol. 18, pp. 499–532, 1989.

[5] R. Cole, "Parallel merge sort," *SIAM J. Comput.*, Vol. 17, pp. 770-785, 1988.

[6] Paul, W., Vishkin, U., and Wagener, H. "Parallel dictionaries on 2-3 trees." *Proc. 10th Coll. on Autom., Lang., and Prog., LNCS 154*, Springer, Berlin, 1983, pp. 597-609.



Figure