

Parallel Algorithms for Searching Monotone Matrices on Coarse Grained Multicomputers

Per-Olof Fjällström

Department of Computer and Information Science
Linköping University
Linköping, Sweden

This work has been submitted for publication elsewhere.
Copyright may then be transferred,
and the present version of the article may be superseded by a revised one.
The WWW page at the URL stated below will contain up-to-date information
about the current version and copyright status of this article. Additional
copyright information is found on the next page of this document.

Linköping University Electronic Press
Linköping, Sweden

<http://www.ep.liu.se/ea/cis/1998/006/>

*Published on June 17, 1998 by
Linköping University Electronic Press
581 83 Linköping, Sweden*

**Linköping Electronic Articles in
Computer and Information Science**
ISSN 1401-9841
Series editor: Erik Sandewall

*©1998 Per-Olof Fjällström
Typeset by the author using L^AT_EX
Formatted using étendu style*

Recommended citation:

*<Author>. <Title>. Linköping Electronic Articles in
Computer and Information Science, Vol. 3(1998): nr 06.
<http://www.ep.liu.se/ea/cis/1998/006/>. June 17, 1998.*

This URL will also contain a link to the author's home page.

*The publishers will keep this article on-line on the Internet
(or its possible replacement network in the future)
for a period of 25 years from the date of publication,
barring exceptional circumstances as described separately.*

*The on-line availability of the article implies
a permanent permission for anyone to read the article on-line,
and to print out single copies of it for personal use.
This permission can not be revoked by subsequent
transfers of copyright. All other uses of the article,
including for making copies for classroom use,
are conditional on the consent of the copyright owner.*

*The publication of the article on the date stated above
included also the production of a limited number of copies
on paper, which were archived in Swedish university libraries
like all other written works published in Sweden.
The publisher has taken technical and administrative measures
to assure that the on-line version of the article will be
permanently accessible using the URL stated above,
unchanged, and permanently equal to the archived printed copies
at least until the expiration of the publication period.*

*For additional information about the Linköping University
Electronic Press and its procedures for publication and for
assurance of document integrity, please refer to
its WWW home page: <http://www.ep.liu.se/>
or by conventional mail to the address stated above.*

Abstract

We present parallel algorithms for geometric problems on coarse grained multicomputers. More specifically, for a square mesh-connected p -processor computer, we show that:

1. The implicit row maxima problem on a totally monotone $n \times n$ matrix can be solved in $O((n/p) \log p)$ time, if $n \geq p^2$.
2. The all-farthest-neighbors problem for a convex n -gon can be solved in $O(n/\sqrt{p})$ time, if $n \geq p^2/4$.
3. The maximum-perimeter triangle inscribed in a convex n -gon can be found in $O(n \log n/\sqrt{p})$ time, if $n \geq p^2$.

The solutions to the two latter problems are based on the reduction of these problems to searching problems in totally monotone matrices.

Keywords: Computational geometry, parallel algorithms, coarse grained multicomputers, totally monotone matrices

The work presented here is funded by CENIIT (the Center for Industrial Information Technology) at Linköping University.

1 Introduction

Given a two-dimensional matrix A , the *row maxima* problem is to decide for each row, the index of the leftmost column containing the maximum value in the row. Let $\hat{c}(r)$ denote the index of the leftmost column containing the maximum value in the r -th row. The matrix A is *monotone* if $r_1 < r_2$ implies that $\hat{c}(r_1) \leq \hat{c}(r_2)$, and A is *totally monotone* if every 2×2 submatrix is monotone.

Totally monotone matrices were introduced by Aggarwal et al. [1], who presented efficient sequential algorithms for solving the row maxima problem in monotone matrices. For example, they showed that given a totally monotone $n \times m$ matrix with $n \leq m$, this problem can be solved in $O(m)$ time. They also showed that several geometric problems can be reduced to the row maxima problem in totally monotone matrices. An example of such a problem is to find a farthest neighbor of each vertex in a given convex n -gon. This problem can be solved in $O(n)$ time by finding row maxima in a totally monotone $n \times (2n - 1)$ matrix.

Aggarwal and Park [2] generalized the notion of totally monotone matrices to multidimensional matrices, and exhibited several geometric problems that can be solved by finding maxima in such matrices. For example, they showed that the problem of finding a maximum-perimeter d -gon inscribed in a given n -gon can be solved by finding a largest entry in a d -dimensional matrix. They solve this problem sequentially in $O(n(d + \log n))$ time, and in time $O((d + \log n) \log n \log \log n)$ using $n / \log \log n$ processors on a CREW PRAM.

Several PRAM algorithms for finding row maxima in totally monotone $n \times m$ matrices have been proposed [2, 3, 4, 5]. In the following, we restrict our attention to deterministic algorithms for the case $n = m$. Aggarwal and Park [2] give an $(\log n \log \log n)$ time and $O(n \log n)$ work CREW algorithm. Atallah and Kosaraju [3] give an $O(\log n)$ time and $O(n \log n)$ work EREW algorithm. Bradford et al. [5] give an EREW algorithm that runs in time $O(\log n \sqrt{\log n \log \log n})$ and does $O(n \sqrt{\log n \log \log n})$ work. Bradford et al. also give work efficient CREW and CRCW algorithms.

Most of the research on parallel algorithms for geometric problems has focused on fine-grain parallel models of computation [6, 7, 8]. It is only during the last couple of years that researchers have designed parallel geometric algorithms for *coarse grained multicomputers* [9, 10, 11, 12, 13, 14, 15, 16, 17]. A coarse grained multicomputer consists of several powerful processors connected by an interconnection network. For this model of computation it is reasonable to assume that the size of each local memory is larger than the number of processors. Many commercially available parallel computers are of this kind.

The main results of this paper are as follows. (The time complexities refer to a square mesh-connected p -processor multicomputer.)

1. The implicit row maxima problem on a totally monotone $n \times n$ matrix can be solved in $O((n/p) \log p)$ time, if $n \geq p^2$. (For each column c , we find all rows r such that $\hat{c}(r) = c$.)

2. The all-farthest-neighbors problem for a convex n -gon can be solved in $O(n/\sqrt{p})$ time, if $n \geq p^2/4$.
3. The maximum-perimeter triangle inscribed in a convex n -gon can be found in $O(n \log n/\sqrt{p})$ time, if $n \geq p^2$.

According to Brent's theorem [18], a PRAM algorithm that runs in time t and does work w can be simulated on a p -processor PRAM to run in time $O(w/p + t)$. We know also that a PRAM algorithm that runs in time t and uses p processors is deterministically simulated in time $\Omega(t\sqrt{p})$ on a p -processor mesh-connected computer [19].

From this follows that our algorithms are faster than the algorithms that would be obtained by simulations of the above PRAM algorithms. For example, by simulating the EREW algorithm of Bradford et al. on a p -processor mesh-connected multicomputer, the all-farthest-neighbors problem can be solved in $O((n/p + \log n)\sqrt{\log n \log \log n \sqrt{p}})$ time. Similarly, by simulating the algorithm of Agarwal and Park, the maximum-perimeter triangle inscribed in a convex n -gon can be found in $O((n/p + \log \log n)\log^2 n \sqrt{p})$ time.

We organize this paper as follows. In Section 2, we describe the basic operations used by our algorithms. In Sections 3 and 4, we introduce the row maxima and plane maxima problems for totally monotone matrices, describe how these problems are related to the geometric problems we consider, and present parallel algorithms for these problems. Section 5 offers some concluding remarks.

2 Basic Operations

Our algorithms use a few basic operations; in the following we describe these operations, and give their time complexities for a square mesh-connected multicomputer with p processors. The operations are assumed to use cut-through routing. The parameters t_s , t_h and t_w denote the start-up time, per-hop time, and per-word transfer time, respectively. For a detailed description and analysis of the operations, see Kumar et al. [20].

1. *Monotone routing*: Each processor $P(i)$ sends at most one m -word message. The destination address, $d(i)$, of the message is such that if both $P(i)$ and $P(i')$, $i < i'$, send messages, then $d(i) < d(i')$. Monotone routing takes $O((t_s + t_w m)\sqrt{p})$ time.
2. *Segmented broadcast*: Processors with indexes $i_1 < i_2 \dots < i_q$, are selected; each processor $P(i_j)$ sends the same m -word message to processors $P(i_j + 1)$ through $P(i_{j+1})$ (but different processors may send different messages). The time complexity is $O((t_s + t_w m) \log p + t_h \sqrt{p})$.
3. *Multinode broadcast*: Every processor sends the same m -word message to every other processor (but different processors may send different messages). Multinode broadcasting requires $O(t_w m p + t_s \sqrt{p})$ time.
4. *Reduction*: Given a sequence a_1, a_2, \dots, a_p of size m vectors with vector a_i stored in processor $P(i)$, and an associative operator \otimes , the reduction

operation computes $s[j] = a_1[j] \otimes \cdots \otimes a_p[j]$, $j = 1, 2, \dots, m$, with $s[j]$ stored in each processor. The time complexity is $O((t_s + t_w m) \log p + t_h \sqrt{p})$. In *segmented* reduction, we apply the reduction operation to disjoint subsequences of consecutive vectors.

3 Two-Dimensional Matrices

In this section we give parallel algorithms for finding row maxima in two-dimensional totally monotone matrices. More specifically, in Section 3.1 we give an algorithm for solving the so-called implicit row maxima problem for an explicitly given matrix. In Section 3.2, we describe an algorithm for the all-farthest-neighbors problem for a convex polygon.

As mentioned in Section 1, Aggarwal et al. [1] showed that the row maxima problem for a totally monotone $n \times m$ matrix can be solved sequentially in time $O(m)$ if $n \leq m$. For monotone matrices, we can also define the *implicit row maxima* problem: for each column c , determine the pair $(l(c), u(c))$ such that for every row r , if $\hat{c}(r) = c$ then $l(c) \leq r \leq u(c)$.

For arbitrary monotone matrices, the implicit row maxima problem can be solved in $O(m \log n)$ time by a straightforward divide-and-conquer algorithm. For totally monotone matrices we can do better. If $n \leq m$, we can easily solve the problem in $O(m)$ time by first solving the row maxima problem. For the case $n > m$, Aggarwal et al. give an algorithm that runs in time $O(m(1 + \log(n/m)))$. This implies that for arbitrary totally monotone matrices, the row maxima problem can always be solved in $O(n + m)$ time.

3.1 Implicit Row Maxima of an $n \times n$ Matrix

In this section we consider the implicit row maxima problem for a totally monotone $n \times n$ matrix A given explicitly. More specifically, each processor $P(k)$ contains A^k , the submatrix of A consisting of columns indexed from $(k-1)n/p+1$ through kn/p . (For simplicity, we assume that n is an integer multiple of p .) At the end of execution, the solution for column c , i.e., $(l(c), u(c))$, is stored in the processor that contains column c . (If no row maxima are found in column c , we set $l(c) > u(c)$.)

Algorithm: 3.1

1. For $0 \leq i \leq n/p$, let $r_i = ip$. Compute $\hat{c}(r_i)$, $1 \leq i \leq n/p$. Let $\hat{c}(r_0) = 1$.
2. Let \hat{c}_i denote $\hat{c}(r_i)$, and let A_i , $1 \leq i \leq n/p$, be the submatrix of A consisting of rows $r_{i-1} + 1$ through $r_i - 1$ and columns \hat{c}_{i-1} through \hat{c}_i . For each submatrix A_i , let k_{i-1} and k_i be the indices such that \hat{c}_{i-1} and \hat{c}_i are columns in $A^{k_{i-1}}$ and A^{k_i} , respectively. See Figure 1.
For each A_i such that $k_{i-1} = k_i$, locally solve the implicit row maxima problem. This solves the implicit row maxima problem for each column c such that $\hat{c}_{i-1} < c < \hat{c}_i$ and $k_{i-1} = k_i$.
3. For each A_i such that $k_{i-1} < k_i$, do as follows.

	A^1	A^2	A^3	A^4
r_1	A_1^1	A_1^2	A_1^3	
r_2			A_2	
r_3			A_3	
r_4			A_4^3, A_4^4	
r_5				A_5
r_6				A_6

Figure 1: Example with $p = 4$ and $n = 24$. For columns in matrices A_2 , A_3 , A_5 , and A_6 , the implicit row maxima problem is solved in Step 2. For remaining columns, the problem is solved in Steps 3 or 4.

- (a) Let A_i^k be the submatrix of A_i with its columns in A^k , $k_{i-1} \leq k \leq k_i$. Locally solve the implicit row maxima problem for A_i^k . Let $(l_i^k(c), u_i^k(c))$ denote the solution for column c in A_i^k .
 - (b) Locally solve the row maxima problem for A_i^k . Let $\hat{c}_i^k(r)$ denote the solution for row r .
 - (c) Solve the row maxima problem for A_i . Let $\hat{c}(r)$ denote the solution for row r .
 - (d) For $r = r_{i-1} + 1, \dots, r_i - 1$ and $k_{i-1} \leq k \leq k_i$, do as follows. If $r > u_i^k(\hat{c}_i^k(r))$ then do nothing. If $\hat{c}_i^k(r) < \hat{c}(r)$, set $u_i^k(\hat{c}_i^k(r)) = r - 1$; otherwise, if $\hat{c}_i^k(r) > \hat{c}(r)$, set $l_i^k(\hat{c}_i^k(r)) = r + 1$.
4. It remains to solve the implicit row maxima problem for each column \hat{c}_i . To do this it is sufficient to combine the solutions for column \hat{c}_i in the matrices A_i and A_{i+1} .

Theorem 1 *The implicit row maxima problem on a totally monotone $n \times n$ matrix can be solved in $O((n/p) \log p)$ time on a square mesh-connected p -processor multicomputer, if $n \geq p^2$.*

Proof. In Step 1, we first locally solve the row maxima problem for the submatrix of A^k consisting of rows $r_1, r_2, \dots, r_{n/p}$. This takes $O(n/p)$ time. Let $\hat{c}^k(i)$, $1 \leq i \leq n/p$, denote the solution. That is, $\hat{c}^k(i)$ is the column in A where the maximum value in row r_i in A^k is found. Next, in each processor $P(k)$ create the records $(A(r_i, \hat{c}^k(i)), \hat{c}^k(i))$. Then, for each value of i , find the record with the largest first component. (If more than one record has the largest first component, the one with smallest second component is selected.) This is achieved by the reduction operation in $O((t_s + t_w n/p) \log p + t_h \sqrt{p})$ time. Finally, set $\hat{c}(r_i)$ equal to the second component of the selected record.

In Step 2, the implicit row maxima problem for A_i is solved in $O((\hat{c}_i - \hat{c}_{i-1} + 1) \log p)$ time by divide-and-conquer. In the worst case (all A_i are part of the same A^k), Step 2 takes time $O((n/p) \log p)$.

The time complexity for Step 3(a) is the same as for Step 2. Steps 3(b) and 3(d) take $O(p + n/p)$ time. The computations in Step 3(c) are done by segmented reduction operations in time $O((t_s + t_w p) \log p + t_h \sqrt{p})$. The last step takes time $O(n/p)$. \square

3.2 All-farthest-neighbors in a convex polygon

Given a convex polygon with vertices p_1, p_2, \dots, p_n (in clockwise order), the *all-farthest-neighbors* problem is to find a farthest neighbor of each vertex. Aggarwal et al. [1] show how to reformulate this problem as a row maxima problem on a totally monotone matrix. More specifically, we can solve the problem by solving the row maxima problem for an $n \times (2n - 1)$ matrix A defined as follows:

$$a_{i,j} = \begin{cases} j - i & \text{if } j \leq i \\ d(p_i, p_{\circ j}) & \text{if } i < j < i + n \\ -1 & \text{otherwise,} \end{cases}$$

where $d(p_i, p_j)$ denotes the Euclidean distance between p_i and p_j , and $\circ j$ denotes $((j - 1) \bmod n) + 1$. To first explicitly construct A and then solve the row maxima problem would obviously take $\Omega(n^2)$ time. Instead, every time a particular entry is needed, we compute it in constant time, and the problem can be solved in $O(n)$ time. In this section, we show how the all-farthest-neighbors problem can be solved on a multicomputer.

To simplify the exposition, we assume that $2n$ is an integer multiple of p , and that p is even. Initially, the vertices are distributed as follows over the processors: processors $P(k)$ and $P(k + p/2)$, $k = 1, 2, \dots, p/2$, contain vertices p_j , $(k - 1)2n/p < j \leq k2n/p$. With processor $P(k)$, $1 \leq k \leq p$, is (implicitly) associated the submatrix A^k of A consisting of columns indexed from $(k - 1)2n/p + 1$ through $k2n/p$. At the end of computation the index of a farthest neighbor of each vertex p_j , $(k - 1)2n/p < j \leq k2n/p$, is stored in processor $P(k)$. For an example, see Figure 2.

Algorithm 3.2:

1. For $0 \leq i \leq p/2$, let $r_i = i2n/p$. Compute $\hat{c}(r_i)$, $1 \leq i \leq p/2$. Let $\hat{c}(r_0) = 1$.
2. For $1 \leq i \leq p/2$, let A_i , k_{i-1} and k_i be defined as in Step 2 of Algorithm 3.1. To solve the row maxima problem for A_i such that $k_{i-1} = k_i$, do as follows.
 - (a) For $k = 1, 2, \dots, p$, determine the integers (if they exist) $f(k) = \min\{i : k_{i-1} = k\}$ and $l(k) = \max\{i : k_i = k\}$. That is, if $f(k) \leq i \leq l(k)$ then A_i lies completely within A^k .
 - (b) For $k = 1, 2, \dots, p$ such that $f(k) \leq l(k)$, copy the vertices in $P(k)$ to processors $P(f(k))$ through $P(l(k))$. Locally solve the row maxima problem for A_i , $l(k) \leq i \leq f(k)$, in the processor $P(i)$.

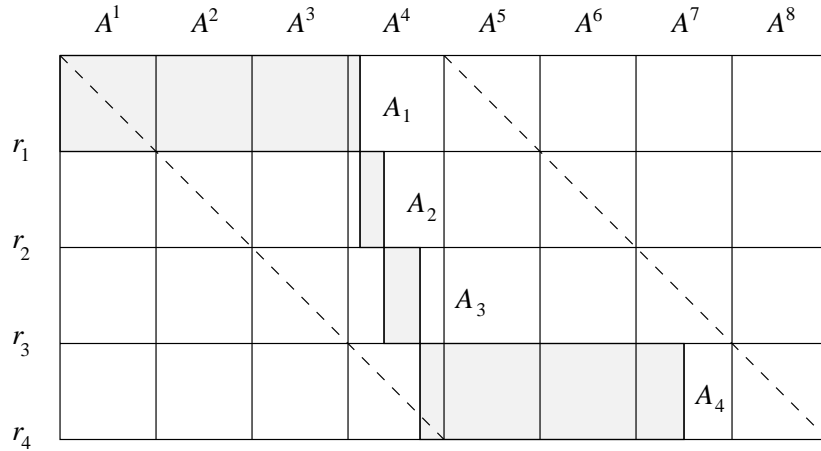


Figure 2: Example with $p = 8$. In Step 2, the processor $P(4)$ sends its vertices to processors $P(2)$ and $P(3)$, which then solve the row maxima problem for A_2 and A_3 , respectively. In Step 3, processors $P(1)$ and $P(4)$ send their vertices to processors $P(2)$ through $P(4)$ and $P(4)$ through $P(7)$, respectively. Then, processors $P(2)$ through $P(4)$ and $P(4)$ through $P(7)$, collectively solve the row maxima problem for A_1 and A_4 , respectively.

3. For each A_i such that $k_{i-1} < k_i$, do as follows.
 - (a) Copy the vertices in $P(i)$ to processors $P(k_{i-1})$ through $P(k_i)$. Let A_i^k be the submatrix of A_i with its columns in A^k , $k_{i-1} \leq k \leq k_i$. Locally solve the row maxima problem for A_i^k .
 - (b) Solve the row maxima problem for A_i . Copy the solution to the processor $P(i)$.

Theorem 2 *The all-farthest-neighbors problem for a convex n -gon can be solved in $O(n/\sqrt{p})$ time on a square mesh-connected p -processor multicomputer, if $n \geq p^2/4$.*

Proof. In Step 1, we first broadcast each vertex p_{r_i} to every processor. Each processor $P(k)$ has now sufficient information to compute entries in row r_i in submatrix A^k . Next, we determine $\hat{c}(r_i)$ using the same techniques as in Step 1 of Algorithm 3.1. That is, with $p/2 \leq 2n/p$, Step 1 can be done in $O(n/p + (t_s + t_w p) \log p + t_h \sqrt{p})$ time.

Step 2(a) takes $O(p)$ time, and the copying in Step 2(b) can be done by monotone routing followed by segmented broadcasting. This takes $O((t_s + t_w n/p + t_h) \sqrt{p})$ time. Since each A_i is a totally monotone $(2n/p - 1) \times m'$, $1 \leq m' \leq 2n/p$, matrix, we can solve the row maxima problem in $O(n/p)$ time. (Note that we do not explicitly compute all of the entries in any submatrix).

In Step 3(a), it takes $O((t_s + t_w n/p + t_h) \sqrt{p})$ time to copy the vertices, and then $O(n/p)$ time to solve the row maxima problems. In Step 3(b), the row maxima are computed in time $O((t_s + t_w n/p) \log p + t_h \sqrt{p})$ using segmented reduction. Finally, the results are copied in $O((t_s + t_w n/p) \sqrt{p})$ time using monotone routing. \square

4 Multidimensional Matrices

Aggarwal and Park [2] generalize the row maxima problem to multidimensional matrices. More specifically, for $d \geq 3$ let $A = \{a_{i_1, i_2, \dots, i_d}\}$ be an $n_1 \times n_2 \times \dots \times n_d$ matrix. The *plane maxima* problem is to determine, for $i_1 = 1, 2, \dots, n_1$, the $(d-1)$ -tuple $(i_2(i_1), i_3(i_1), \dots, i_d(i_1))$ such that

$$a_{i_1, i_2(i_1), \dots, i_d(i_1)} = \max_{i_2, \dots, i_d} a_{i_1, i_2, \dots, i_d}.$$

(If a plane contains several maxima, the first in lexicographic order by second through d -th indices is chosen.) Besides the plane maxima problem, Aggarwal and Park consider the *tube maxima* problem, in which the location of a maximum entry is to be determined for every pair (i_1, i_2) of first and second indices.

In analogy to the two-dimensional case, A is monotone if $i_1 > i'_1$ implies that $i_k(i_1) \geq i_k(i'_1)$ for $k = 2, 3, \dots, d$. Moreover, A is totally monotone if every $2 \times 2 \times \dots \times 2$ d -dimensional submatrix is monotone, and every plane (the $(d-1)$ -dimensional submatrix corresponding to a fixed value of the first index) is totally monotone.

Aggarwal and Park give efficient sequential and parallel algorithms for the plane and tube maxima problems on totally monotone matrices. More specifically, the plane maxima of an $n_1 \times n_2 \times \dots \times n_d$ d -dimensional totally monotone matrix can be computed sequentially in $O((n_{d-1} + n_d) \prod_{k=1}^{d-2} \log n_k)$ time. Moreover, the plane maxima problem on a totally monotone $n \times n \times n$ matrix can be solved in time $O(\log^2 n \log \log n)$ using $n / \log \log n$ processors on a CREW PRAM. Aggarwal and Park show also that several geometric and other kinds of problems can be reduced to the plane or tube maxima problem on totally monotone matrices. In the next section, we present one of these problems, and give a parallel algorithm for it.

4.1 Maximum-Perimeter Inscribed Triangles

Given a convex polygon P with vertices p_1, p_2, \dots, p_n (in clockwise order), the *maximum-perimeter inscribed triangle* problem is to find a triangle T contained in P with a maximum perimeter. Since the vertices of T must be vertices of P , it suffices to determine the triple (i_1, i_2, i_3) , $1 \leq i_1 < i_2 < i_3 \leq n$, such that $per(i_1, i_2, i_3)$, the perimeter of the triangle with vertices p_{i_1} , p_{i_2} and p_{i_3} , is maximum. Consider the $(n-2) \times (n-1) \times n$ matrix A where

$$a_{i_1, i_2, i_3} = \begin{cases} -\infty & \text{if } i_2 \leq i_1 \\ per(i_1, i_2, i_3) & \text{if } i_1 < i_2 < i_3 \\ i_3 - i_2 & \text{otherwise.} \end{cases}$$

By using the quadrangle inequality (i.e., for any convex quadrangle, the sum of the diagonals is greater than the sum of opposite sides), one can show that every plane of A is totally monotone, and that A is monotone. This implies that a maximum entry in A can be found sequentially in $O(n \log n)$ time by divide-and-conquer. In the following, we show how this problem can be solved on a multicomputer.

Our algorithm is essentially a parallelization of the corresponding sequential divide-and-conquer algorithm. The main challenge is to ensure that a processor has local access to the appropriate vertices while avoiding that it receives too many vertices. To simplify the exposition, we assume that both n and p are integer powers of two. In our description of the algorithm we refer to various submatrices of matrix A . However, as in Algorithm 3.2, whenever a particular matrix entry is needed it is computed in constant time. Let $I(k) = \{i : (k-1)n/p < i \leq kn/p\}$. We define the k -th *block of planes* as the submatrix of A consisting of all entries with index $i_1 \in I(k)$. Similarly, the k -th block of rows (columns) consists of all entries with index $i_2 \in I(k)$ ($i_3 \in I(k)$). Initially, each processor $P(k)$ contains vertices $p_i, i \in I(k)$.

We divide the computations into two parts. Let $i_1(i) = ip$. In Part I, we decide the maxima for planes $i_1(i), i = 1, 2, \dots, n/p - 1$. That is, for each value of i we compute $(i_2(i_1(i)), i_3(i_1(i)))$ and the value of the corresponding entry. The first step of Part I is that each processor $P(k)$ broadcasts vertex $p_{kn/p}$ to all processors. The rest of Part I is divided into $\log(n/p)$ phases. In the s -th phase, $s = 1, 2, \dots, \log(n/p)$, we determine the maxima for planes $i_1(i) = ip$, where $i = (2l-1)(n/p)/2^s$ and $l = 1, 2, \dots, 2^{s-1}$.

Let $A(i), i = 1, 2, \dots, n/p$, be the submatrices of A such that $A(i) = \{a_{i_1, i_2, i_3}\}$, where $i_1(i-1) < i_1 < i_1(i)$, and $i_j(i_1(i-1)) \leq i_j \leq i_j(i_1(i))$ for $j = 2, 3$. (We define $i_2(i_1(0)) = 1, i_3(i_1(0)) = 2, i_2(i_1(n/p)) = n-1$ and $i_3(i_1(n/p)) = n$.) In Part II, we find maximum entries in these matrices. More specifically, Part II consists of $\log p$ phases, and in the first phase we compute maximum entries in the matrices that are completely contained in a block of rows or a block of columns. Each of the remaining matrices is decomposed into two submatrices, which then are processed in the next phase.

We continue by describing how the s -th phase of Part I is done. At the beginning of the s -th phase each processor has the results from previous phases in its local memory. By the properties of A , we know that the maximum for plane $i_1(i) = ip$, where $i = (2l-1)(n/p)/2^s$ and $l = 1, 2, \dots, 2^{s-1}$, lies within the submatrix $B(i) = \{a_{i_1(i), i_2, i_3}\}$, where $i_j^l(i) \leq i_j \leq i_j^u(i)$ for $j = 2, 3$, and $i_j^l(i)$ and $i_j^u(i)$ are locations of plane maxima computed in previous phases. More specifically, $i_j^l(i) = i_j(i_1(i - (n/p)/2^s))$ and $i_j^u(i) = i_j(i_1(i + (n/p)/2^s))$. In the s -th phase, we first solve the row maxima problem for each submatrix $B(i)$, and then decide, for each submatrix, which of its row maxima is the largest.

Algorithm 4.1: (s -th phase of Part I)

1. Broadcast vertices $p_{i_1(i)}$, where $i = (2l-1)(n/p)/2^s$ and $l = 1, 2, \dots, 2^{s-1}$, to all processors.
2. For each matrix $B(i), i = (2l-1)(n/p)/2^s$ and $l = 1, 2, \dots, 2^{s-1}$, and each integer k such that $i_2^l(i) \leq kn/p < i_2^u(i)$, find the largest entry in row kn/p . That is, determine $i_3(i, k)$ such that

$$a_{i_1(i), kn/p, i_3(i, k)} = \max_{i_3^l(i) \leq i_3 \leq i_3^u(i)} a_{i_1(i), kn/p, i_3}.$$

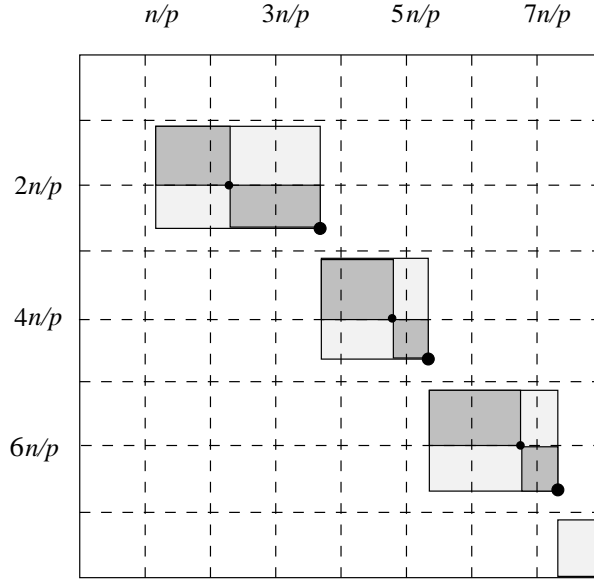


Figure 3: Example with $p = 8$ and $n \geq 64$. The maxima for planes $i_1(i) = n/4, n/2, 3n/4$, are shown by large bullets. Larger shaded regions correspond to the matrices $B(i)$, $i = n/64, 3n/64, 5n/64, 7n/64$. Smaller bullets correspond to the row maxima computed in Step 2. The darker shaded regions represent matrices replacing matrices $B(i)$.

3. Using the row maxima computed in the previous step, replace matrix $B(i)$ by submatrices. More specifically, if $i_2^l(i) \leq kn/p < i_2^u(i)$ for $k = l(i), l(i) + 1, \dots, u(i)$, replace $B(i)$ by submatrices $B(i, j)$, $j = l(i), l(i) + 1, \dots, u(i) + 1$, such that $B(i, j) = \{a_{i_1(i), i_2, i_3}\}$, where

$$\max(i_2^l(i), (j-1)n/p + 1) \leq i_2 \leq \min(i_2^u(i), jn/p - 1),$$

$$\max(i_3^l(i), i_3(i, j-1)) \leq i_3 \leq \min(i_3^u(i), i_3(i, j)).$$

See Figure 3.

4. For $i = 1, 2, \dots$, let $C_i = \{a_{i_1^l(i), i_2, i_3}\}$, where $i_j^l(i) \leq i_j \leq i_j^u(i)$ for $j = 2, 3$, be the ordered collection of matrices remaining after the previous step. That is, $i < i'$ implies that $i_j^u(i) \leq i_j^l(i')$ for $j = 2, 3$. (We define $i_1^l(i) = i_1(j)$ if C_i has replaced (or is identical to) $B(j)$.)

For each matrix C_i let $k_j^l(i)$ and $k_j^u(i)$ be integers such that $i_j^l(i) \in I(k_j^l(i))$ and $i_j^u(i) \in I(k_j^u(i))$. To solve the row maxima problem for each matrix C_i , continue as follows.

- (a) For $k = 1, 2, \dots, p$, and $j = 2, 3$, determine the integers

$$f_j(k) = \min\{i : k_j^l(i) = k\}, \quad l_j(k) = \max\{i : k_j^u(i) = k\},$$

$$f'_j(k) = \max\{i : k_j^u(i) < k\} \text{ and } l'_j(k) = \min\{i : k_j^l(i) > k\}.$$

That is, if $f_2(k) \leq i \leq l_2(k)$, then C_i is completely contained in the k -th block of rows. Moreover, $f_2'(k)$ is the index of the highest-indexed matrix that lies completely above of the k -th block of rows.

- (b) For $k = 1, 2, \dots, p$ such that $f_3(k) \leq l_3(k)$, unless $k_2^u(f_3'(k)) = k_2^l(l_3'(k))$, do:
- i. Copy vertices $p_j, j \in I(k)$, to processors $P(k_2^l(f_3(k)))$ through $P(k_2^l(l_3(k)))$.
 - ii. Locally solve the row maxima problem for $C_i, f_3(k) \leq i \leq l_3(k)$, in the processor $P(k_2^l(i))$.
- (c) For $k = 1, 2, \dots, p$ such that $f_2(k) \leq l_2(k)$, unless $k_3^u(f_2'(k)) = k_3^l(l_2'(k))$, do:
- i. Copy vertices $p_j, j \in I(k)$, to processors $P(k_3^l(f_2(k)))$ through $P(k_3^u(l_2(k)))$.
 - ii. Locally solve the row maxima for $C_i, f_2(k) \leq i \leq l_2(k)$ and $k_3^l(i) = k_3^u(i)$, in the processor $P(k_3^l(i))$.
 - iii. Solve the row maxima problem for $C_i, f_2(k) \leq i \leq l_2(k)$ and $k_3^l(i) < k_3^u(i)$, as in Step 3 of Algorithm 3.2.

5. Find the maximum entry in each submatrix $B(i)$.

Lemma 1 *The s -th phase of Part I takes $O(n/\sqrt{p})$ time on a square mesh-connected p -processor multicomputer, if $n/p \geq p$.*

Proof. In Step 1, no processor broadcasts more than $O(2^{s-1}/p)$ vertices; this takes $O(2^{s-1})$ time. In Step 2, we solve the row maxima problem for a totally monotone matrix B defined as follows. Let $t = (k_1, k_2, \dots)$ be the increasing sequence of integers such that $k \in t$ if and only if $i_2^l(i) \leq kn/p < i_2^u(i)$ for some matrix $B(i)$. Matrix B has n columns and one row for each element in t . More specifically, the entries of B are as follows:

$$b_{i_2, i_3} = \begin{cases} i_3 - \max(i_3^l(i), k_{i_2} n/p) & \text{if } i_3 < \max(i_3^l(i), k_{i_2} n/p) \\ a_{i_1(i), k_{i_2} n/p, i_3} & \text{if } \max(i_3^l(i), k_{i_2} n/p) \leq i_3 \leq i_3^u(i) \\ -\infty & \text{otherwise,} \end{cases}$$

where i is such that $i_2^l(i) \leq k_{i_2} n/p < i_2^u(i)$. The row maxima problem for B can be solved using the same techniques (and within the same amount of time) as in Step 1 of Algorithm 3.2.

Steps 3 and 4(a) take $O(n/p)$ time. Observe that in Steps 4(b)-(c), a single processor may have to solve the row maxima problem for several matrices. However, since the total number of rows and columns of these matrices is $O(n/p)$, these row maxima problems can be solved in $O(n/p)$ time. The requirement in Step 4(b) that we must not have $k_3^u(f_2'(k)) = k_3^l(l_2'(k))$ (and the similar requirement in Step 4(c)), guarantees that no processor receives vertices from more than one processor. Therefore, by using monotone routing followed by segmented broadcasting, we can copy the vertices in $O(n/\sqrt{p})$ time.

It is easy to see that the row maxima problem for a matrix C_i remains unsolved after Step 4(b) only if (1) $k_3^l(i) < k_3^u(i)$, or (2) $k_3^l(i) = k_3^u(i) = k'$

and $k_2^u(f_3^l(k')) = k_2^l(l_3^l(k'))$. Case (1) is taken care of in Step 4(c)(iii). In case (2) we cannot have $k_3^u(f_2^l(\bar{k})) = k_3^l(l_2^l(\bar{k}))$, where $\bar{k} = k_2^l(i)$. Therefore, the row maxima of a matrix C_i are determined either in Step 4(b) or in Step 4(c).

Steps 2 and 4 together compute the row maxima of each matrix $B(i)$. The total number of row maxima is $O(n)$ and each processor computes $O(n/p)$ row maxima. In Step 5 each processor first inspects its row maxima, and finds a (locally) largest row maxima for each $B(i)$. Then, by using the reduction operation a maximum entry in each $B(i)$ is found. Thus, Step 5 takes $O(n/p + (t_s + t_w 2^{s-1}) \log p + t_h \sqrt{p})$ time. \square

As already mentioned, Part II consists of $\log p$ phases. The input to each phase consists of an ordered set of matrices $A(1), A(2), \dots$, each of which is a submatrix of A . That is, $A(i) = \{a_{i_1, i_2, i_3}\}$, where $i_j^l(i) \leq i_j \leq i_j^u(i)$ for $j = 1, 2, 3$, and $i_j^u(i) \leq i_j^l(i')$ if $i < i'$. Moreover, each processor has $i_j^l(i)$ and $i_j^u(i)$ in its local memory. The number of matrices is n/p in the first phase and $O(p)$ in the remaining phases. For matrix $A(i)$ and $j = 1, 2, 3$, let $k_j^l(i)$ and $k_j^u(i)$ be the integers such that $i_j^l(i) \in I(k_j^l(i))$ and $i_j^u(i) \in I(k_j^u(i))$. Observe that $k_1^l(i) = k_1^u(i)$ for each $A(i)$.

We divide the input to each phase into two subsets, M_1 and M_2 , where M_1 consists of matrices $A(i)$ such that $k_2^l(i) = k_2^u(i)$ or $k_3^l(i) = k_3^u(i)$. For each member of M_1 , we determine its maximum entry during the phase. For each matrix of M_2 , we select its middle plane, determine the maximum entry for this plane, and use this entry to replace the matrix by two new submatrices. The set of new submatrices is the input to the next phase.

Algorithm 4.2: (s -th phase of Part II)

1. For $k = 1, 2, \dots, p$, and $j = 1, 2, 3$, determine the integers

$$f_j(k) = \min\{i : k_j^l(i) = k\}, \quad l_j(k) = \max\{i : k_j^u(i) = k\},$$

$$f_j^l(k) = \max\{i : k_j^u(i) < k\} \text{ and } l_j^l(k) = \min\{i : k_j^l(i) > k\}.$$
2. For $k = 1, 2, \dots, p$ such that $f_1(k) \leq l_1(k)$, unless $k_3^u(f_1(k)) = k_3^l(l_1(k))$, copy vertices $p_j, j \in I(k)$, to processors $P(k_3^l(f_1(k)))$ through $P(k_3^u(l_1(k)))$.
3. For $k = 1, 2, \dots, p$ such that $f_2(k) \leq l_2(k)$, unless $k_3^u(f_2(k)) = k_3^l(l_2(k))$, do:

- (a) Copy vertices $p_j, j \in I(k)$, to processors $P(k_3^l(f_2(k)))$ through $P(k_3^u(l_2(k)))$.
- (b) For each $i = f_2(k), \dots, l_2(k)$, unless $k_3^u(f_1^l(\hat{k})) = k_3^l(l_1^l(\hat{k}))$, where $\hat{k} = k_1^l(i)$, find a maximum element in $A(i)$.

More specifically, for $A(i)$ such that $k_3^l(i) = k_3^u(i)$, solve the problem locally in processor $P(k_3^l(i))$. Otherwise, divide $A(i)$ into submatrices $A_l(i), k_3^l(i) \leq l \leq k_3^u(i)$, such that $A_l(i) = \{a_{i_1, i_2, i_3}\}$, $i_j^l(i) \leq i_j \leq i_j^u(i)$ for $j = 1, 2$, and

$$\max\{i_3^l(i), (l-1)n/p + 1\} \leq i_3 \leq \min\{i_3^u(i), ln/p\}.$$

Compute a maximum element of $A_l(i)$ in processor $P(l)$. Finally, find a maximum element of $A(i)$.

4. For $k = 1, 2, \dots, p$ such that $f_1(k) \leq l_1(k)$, unless $k_2^u(f_1(k)) = k_2^l(l_1(k))$, copy vertices $p_j, j \in I(k)$, to processors $P(k_2^l(f_1(k)))$ through $P(k_2^u(l_1(k)))$.
5. For $k = 1, 2, \dots, p$ such that $f_3(k) \leq l_3(k)$, unless $k_2^u(f_3(k)) = k_2^l(l_3(k))$, do:
 - (a) Copy vertices $p_j, j \in I(k)$, to processors $P(k_2^l(f_3(k)))$ through $P(k_2^u(l_3(k)))$.
 - (b) For each $i = f_3(k), \dots, l_3(k)$, unless $k_2^u(f_1(\hat{k})) = k_2^l(l_1(\hat{k}))$, where $\hat{k} = k_1^l(i)$, find a maximum element in $A(i)$. This can be done similarly to Step 3(b).
6. For $k = 1, 2, \dots, p$, determine the integers
$$f_3''(k) = \min\{\hat{k} : k_3^u(f_1(\hat{k})) = k_3^l(l_1(\hat{k})) = k \text{ and } k_2^u(f_1(\hat{k})) = k_2^l(l_1(\hat{k}))\},$$

$$l_3''(k) = \max\{\hat{k} : k_3^u(f_1(\hat{k})) = k_3^l(l_1(\hat{k})) = k \text{ and } k_2^u(f_1(\hat{k})) = k_2^l(l_1(\hat{k}))\},$$

$$f_2''(k) = \min\{\hat{k} : k_2^u(f_1(\hat{k})) = k_2^l(l_1(\hat{k})) = k \text{ and } k_3^u(f_1(\hat{k})) = k_3^l(l_1(\hat{k}))\},$$

$$l_2''(k) = \max\{\hat{k} : k_2^u(f_1(\hat{k})) = k_2^l(l_1(\hat{k})) = k \text{ and } k_3^u(f_1(\hat{k})) = k_3^l(l_1(\hat{k}))\}.$$
7. For $k = 1, 2, \dots, p$, do as follows.
 - (a) If $f_3''(k) \leq l_3''(k)$, copy vertices $p_j, j \in I(k)$, to processors $P(f_3''(k))$ through $P(l_3''(k))$.
 - (b) If $f_2''(k) \leq l_2''(k)$ exists, copy vertices $p_j, j \in I(k)$, to processors $P(f_2''(k))$ through $P(l_2''(k))$.
 - (c) If $k_3^u(f_1(k)) = k_3^l(l_1(k))$ and $k_2^u(f_1(k)) = k_2^l(l_1(k))$, locally (i.e., in processor $P(k)$) compute a maximum element of each $A(i), f_1(k) \leq i \leq l_1(k)$.
8. For each $A(i)$ such that $k_2^l(i) < k_2^u(i)$ and $k_3^l(i) < k_3^u(i)$, do as follows.
 - (a) Select a median plane $m(i)$ in $A(i)$, and broadcast vertex $p_{m(i)}$ to every processor.
 - (b) Determine a maximum entry for plane $m(i)$.
 - (c) Unless $m(i)$ is the only plane in $A(i)$, replace $A(i)$ by the two submatrices of $A(i)$ consisting of planes lying below or above $m(i)$.

Lemma 2 *The s -th phase of Part II takes $O(n/\sqrt{p})$ time on a square mesh-connected p -processor multicomputer, if $n/p \geq p$.*

Proof. Steps 1 and 6 take $O(n/p)$ time. The copying of vertices in Steps 2-5 and 7 can be done in $O(n/\sqrt{p})$ time. In Steps 3, 5 and 7, each processor first solves the plane maxima problem in the matrices for which it is responsible. Then, the largest plane maximum is found. Note that although a single processor may be responsible for several matrices, the total number of planes, rows and columns in these matrices is $O(n/p)$. Therefore, these computations take $O((n/p) \log(p/2^{s-1}))$ time.

Consider a matrix $A(i)$ such that $k_2^l(i) = k_2^u(i) = \bar{k}$ and $k_3^l(i) < k_3^u(i)$. Let $\hat{k} = k_1^l(i)$. Then neither $k_3^u(f_1(\hat{k})) = k_3^l(l_1(\hat{k}))$ nor $k_3^u(f_2(\bar{k})) = k_3^l(l_2(\bar{k}))$

can be true. Therefore, a maximum entry for $A(i)$ is computed in Step 3(b). Similarly, a maximum entry for a matrix $A(i)$ such that $k_3^l(i) = k_3^u(i) = k'$ and $k_2^l(i) < k_2^u(i)$ is computed in Step 5(b).

Next, consider a matrix $A(i)$ such that $k_2^l(i) = k_2^u(i) = \bar{k}$, $k_3^l(i) = k_3^u(i) = k'$ and $\hat{k} = k_1^l(i)$. Suppose that both $k_3^u(f_1'(\hat{k})) = k_3^l(l_1'(\hat{k}))$ and $k_2^u(f_1'(\hat{k})) = k_2^l(l_1'(\hat{k}))$ are false. Since $k_3^u(f_2'(\bar{k})) = k_3^l(l_2'(\bar{k}))$ and $k_2^u(f_3'(k')) = k_2^l(l_3'(k'))$ cannot both be true, a maximum entry in $A(i)$ is computed in Steps 3(b) or 5(b). Suppose next that $k_3^u(f_1'(\hat{k})) = k_3^l(l_1'(\hat{k}))$ but $k_2^u(f_1'(\hat{k})) = k_2^l(l_1'(\hat{k}))$ is false. Then $k_2^u(f_3'(k')) = k_2^l(l_3'(k'))$ must be false. Therefore, a maximum entry in $A(i)$ is computed in Step 5(b). Conversely, if $k_3^u(f_1'(\hat{k})) = k_3^l(l_1'(\hat{k}))$ is false but $k_2^u(f_1'(\hat{k})) = k_2^l(l_1'(\hat{k}))$ is true, then $k_3^u(f_2'(\bar{k})) = k_3^l(l_2'(\bar{k}))$ must be false, and a maximum entry in $A(i)$ is computed in Step 3(b). Finally, suppose that both $k_3^u(f_1'(\hat{k})) = k_3^l(l_1'(\hat{k}))$ and $k_2^u(f_1'(\hat{k})) = k_2^l(l_1'(\hat{k}))$ are true. Then a maximum entry in $A(i)$ is computed in Step 7(c).

Since the total number of vertices broadcasted in Step 8(a) is at most $p - 1$, Step 8(a) can be done in $O(p)$ time. The rest of Step 8 is similar to the computations in Algorithm 4.1 and takes $O(n/\sqrt{p})$ time. \square

We have not explicitly described how a maximum element of matrix A is found. Observe, that the largest plane maxima from Part I can be determined in $O(n/p)$ time by local computation. Similarly, the largest of the (at most $p - 1$) plane maxima computed in Step 8 of Algorithm 4.2 can also be determined locally. As for the submatrix maxima computed in Steps 3, 5 and 7 of Algorithm 4.2, it is easy to verify that no processor computes more than $O(n/p)$ plane maxima per phase. This implies that:

Theorem 3 *The maximum-perimeter inscribed triangle problem can be solved in $O(n \log n/\sqrt{p})$ time on a square mesh-connected p -processor multicomputer, if $n/p \geq p$.*

5 Concluding remarks

Although the algorithms that we have presented in this paper differ significantly in their complexity, they all follow the same basic pattern. In the first part of the algorithm, maxima in selected rows or planes are computed. In Algorithms 3.1 and 3.2, the row maxima are computed simultaneously. In the algorithm for finding a maximum-perimeter inscribed triangle, the first part is divided into $O(\log(n/p))$ phases. Using these maxima, the original matrix is then subdivided into submatrices. In the second part, we solve (more or less directly) the (implicit) row or plane maxima problem for the submatrices.

In spite of the simplicity of this pattern, the algorithms (in particular Algorithms 4.1 and 4.2) are rather complicated. Before a processor can find maxima in a submatrix, we must ensure that all vertices required to compute its entries are stored in the processor. On the other hand, we do not want a processor to store more than $O(n/p)$ vertices. Thus, complications arise since we must ensure not only that a processor has the appropriate vertices, but also that it does not receive vertices from several processors simultaneously.

References

- [1] A. Aggarwal, M.M Klawe, S. Moran, P. Shor, and R. Wilber. Geometric applications of a matrix searching algorithm. *Algorithmica*, 2:209–233, 1987.
- [2] A. Aggarwal and J. Park. Notes on searching in multidimensional monotone arrays. In *Proc. 29th Annual IEEE Symposium on Foundations of Computer Science*, pages 497–512, 1988.
- [3] M.J. Atallah and S.R. Kosaraju. An efficient parallel algorithm for the row minima of a totally monotone matrix. In *Proc. Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 394–403, 1991.
- [4] R. Raman and U. Vishkin. Optimal randomized parallel algorithms for computing the row minima of a totally monotone matrix. In *Proc. 5th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'94)*, pages 613–621, 1994.
- [5] P.G. Bradford, R. Fleischer, and M. Smid. More efficient parallel totally monotone matrix searching. *Journal of Algorithms*, 23:386–400, 1997.
- [6] A. Aggarwal, B. Chazelle, L. Guibas, and C. O'Dunlaing. Parallel computational geometry. *Algorithmica*, 3:293–327, 1988.
- [7] M.J. Atallah. Parallel techniques for computational geometry. *Proc. IEEE*, 80(9):1435–1448, 1992.
- [8] S.G. Akl and K.A. Lyons. *Parallel Computational Geometry*. Prentice-Hall, 1993.
- [9] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *Proc. 9th Annual ACM Symposium on Computational Geometry*, pages 298–307, 1993.
- [10] O. Devillers and A. Fabri. Scalable algorithms for bichromatic line segment intersection problems on coarse grained multicomputers. In *Algorithms and Data Structures. Third Workshop, WADS'93*, pages 277–288, 1993.
- [11] X. Deng. A convex hull algorithm on coarse grained multiprocessors. In *Proc. 5th Annual International Symposium on Algorithms and Computation (ISAAC 94)*, pages 634–642, 1994.
- [12] F. Dehne, C. Kenyon, and A. Fabri. Scalable and architecture independent parallel geometric algorithms with high probability optimal time. In *Proc. 6th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 586–593, 1994.
- [13] F. Dehne, X. Deng, P. Dymond, A. Fabri, and A.A. Khokhar. A randomized parallel 3D convex hull algorithm for coarse grained multicomputers. In *Proc. 7th ACM Symposium on Parallel Algorithms and Architectures*, pages 27–33, 1995.

- [14] I. Al-furaih, S. Aluru, S. Goil, and S. Ranka. Parallel construction of multidimensional binary search trees. In *Proc. International Conference on Supercomputing (ICS'96)*, 1996.
- [15] P-O. Fjällström. Parallel interval-cover algorithms for coarse grained multicomputers. Technical Report LiTH-IDA-R-96-39, Dep. of Computer and Information Science, Linköping University, 1996.
- [16] A. Ferreira, C. Kenyon, A. Rau-Chaplin, and S. Ubeda. d -Dimensional range search on multicomputers. Technical Report 96-23, Laboratoire de l'Informatique du Parallelisme, Ecole Normale Supérieure de Lyon, 1996.
- [17] P-O. Fjällström. Parallel algorithms for batched range searching on coarse-grained multicomputers. In *Proc. 11th Annual Int. Symp. on High Performance Computing Systems (HPCS'97)*, pages 167–178, 1997.
- [18] A. Gibbons and W. Rytter. *Efficient parallel algorithms*. Cambridge University Press, 1988.
- [19] A. Pietracaprina, G. Pucci, and J.F. Sibeyn. Constructive deterministic PRAM simulation on a mesh-connected computer. In *6th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'94)*, pages 248–61, 1994.
- [20] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing*. The Benjamin/Cummings Publishing Company, Inc., 1994.