Parallel Output Sensitive Algorithms for Combinatorial and Linear Algebra Problems

John H. Reif [1]

Department of Computer Science, Box 90129, Duke University, Durham, NC 27708-0129

E-mail: reif@cs.duke.edu

This paper gives *output sensitive* parallel algorithms whose performance depends on the output size and are significantly more efficient than previous algorithms for problems with sufficiently small output size. Inputs are $n \times n$ matrices over a fixed ground field. Let $P(n)$ and $M(n)$ be the PRAM processor bounds for $O(\log n)$ time multiplication of two degree n polynomials, and $n \times n$ matrices, respectively. Let $T(n)$ be the time bounds, using $M(n)$ processors, for testing if an $n \times n$ matrix is nonsingular, and if so, computing its inverse. We compute the rank R of a matrix in randomized parallel time $O(\log n + T(R) \log R)$ using $nP(n) + M(R)$ processors ($P(n) + RP(R)$ processors for constant displacement rank matrices, e.g. Toeplitz matrices). We find a maximum linearly independent subset (MLIS) of an n-set of n-dimensional vectors in time $O(T(n) \log n)$ using $M(n)$ randomized processors and we also give output sensitive algorithms for this problem. Applications include output sensitive algorithms for finding: (i) a size R maximum matching in an n-vertex graph using time $O(T(R) \log n)$ and $nP(n)/T(R) + R\, M(R)$ processors, and (ii) a maximum matching in an n-vertex bipartite graph, with vertex subsets of sizes $n_1 \leq n_2$, using time $O(T(n_1) \log n)$ and $nP(n)/T(n_1) + n_1 M(n_1)$ processors.

Keywords: parallel algorithms, randomized algorithms, linear systems, maximum linear independent subset, matrix rank, structured matrices, Toeplitz matrices, displacement rank, output sensitive, bipartite matching.

List of symbols:
O —'oh'
0 — 'zero'
ℓ — 'ell'
1 — 'one'
κ — 'kappa'
K — 'upper case kay'
k — 'lower case kay'
$\times$ — 'times'
n — 'en'

$\mathcal{F}$ — ' cal F'
$\mathcal{NC}$ — ' cal NC'
$\mathcal{RNC}$ — ' cal RNC'
δ — 'delta'
π — 'pi'
Σ — 'upper case Sigma'
Ω — 'upper case Omega'

1. INTRODUCTION

1.1. Assumptions and Preliminary Definitions

Definition of $M(n), P(n)$ and $T(n)$: We assume a fixed ground field $\mathcal{F}$. Our complexity bounds (time and number of processors) are stated for the arithmetic randomized PRAM model, where in one time step each processor may execute a test for equality, perform an operation $(+, -, \times, \div)$ over the field $\mathcal{F}$, or draw a random element from (a specified subset of) $\mathcal{F}$. Recall that in the abstract, we define $M(n)$ to be the number of arithmetic processors for multiplying two matrices of size $n \times n$ in $O(\log n)$ time; currently, $M(n)$ is known to be $O(n^{2.376})$, see [12]. $P(n)$ denotes the number of arithmetic processors for multiplying two degree n polynomials in $O(\log n)$ parallel time. It is known that $P(n) = O(n)$ if the field $\mathcal{F}$ supports an FFT of size n but is otherwise $P(n) = O(n \log \log n)$ [9]. Also, recall that $T(n)$ is the time bound, using $M(n)$ processors, for testing if an $n \times n$ matrix nonsingular, and if nonsingular, computing its inverse. The best known bounds for $T(n)$ depend on the characteristic $char(\mathcal{F})$ and cardinality $card(\mathcal{F})$ of $\mathcal{F}$; they are ([23]) $T(n) = O(\log^2 n)$ if $char(\mathcal{F}) = 0$ or $char(\mathcal{F}) \geq n$, and otherwise ([24]) $T(n)$ remains polylog (in particular, for $0 < char(\mathcal{F}) < n$, $T(n) = O(\log^4 n)/\log char(\mathcal{F})$ if $card(\mathcal{F}) \geq n$, and otherwise $T(n) = O(\log^5 n)/(\log char(\mathcal{F}) \log card(\mathcal{F})))$.

Parallel Complexity Classes and Bounds. $\mathcal{NC}$ is the class of problems solvable in polylog time with a polynomial number of deterministic processors and $\mathcal{RNC}$ is the class of problems solvable in polylog time with a polynomial number of randomized processors. The simplest versions of our randomized algorithms are usually Monte Carlo algorithms with a one-sided failure probability of $1/c$, where c is a constant. To improve the failure probability to $1/n^k$, where k is a constant, we run $(k \log_c n)$ replicates of the parallel algorithm; the overall computation fails iff all the replicates fail; although the processor requirements increase by a factor of $(k \log_c n)$, the parallel time remains the same.

For some of our parallel algorithms, the processor requirements are stated using Brent's slowdown theorem: for a slowdown $s \geq 1$, a parallel algorithm running in time T with P processors runs in time $\lceil sT \rceil$ with $\lceil P/s \rceil$ processors.

Frequently, we will need to solve a problem by sequentially executing two parallel algorithms, the first with time T_1 and processor bound P_1, the second with time T_2 and processor bound P_2. It is obvious that if $T_1 \leq T_2$, then we can, by a factor of T_2/T_1, slow down the first algorithm to T_2 time and thus decrease the processor

[1]Supported by Grants NSF/DARPA CCR-9725021, CCR-96-33567, NSF IRI-9619647, and ONR contract N00014-99-1-0406.

use by the first algorithm by a factor of $\lceil T_1/T_2 \rceil$. Otherwise, if $T_2 \leq T_1$, then we can do a similar slowdown of the second algorithm.

PROPOSITION 1.1. *We can sequentially execute two parallel algorithms, the first with time T_1 and processor bound P_1, the second with time T_2 and processor bound P_2, by appropriate slowdown, in time $T_1 + T_2$ and $\pi_{T_1,T_2}(P_1, P_2)$ processors, where $\pi_{T_1,T_2}(P_1, P_2) = P_1(\lceil T_1/T_2 \rceil) + P_2$ if $T_1 \leq T_2$, and otherwise $\pi_{T_1,T_2}(P_1, P_2) = P_1 + P_2(\lceil T_2/T_1 \rceil)$.*

Note that $\pi_{T_1,T_2}(P_1, P_2) \leq P_1 + P_2$, so this slowdown never yields an increased processor bound; for our results, the time bounds are of the form $T_1 = O(\log n)$ and $T_2 = O(\log^c R)$. Note that if $R = O(1)$, then the reduced processor bounds after constant slowdown, given by $\pi_{T_1,T_2}(P_1, P_2)$, are $P_1 + P_2/\log n$, and if $R = O(n^\epsilon)$, for $\epsilon > 0$, then the reduced processor bounds are $P_1/(\log^{c-1} n) + P_2$. Thus the improvement given by $\pi_{T_1,T_2}(P_1, P_2)$ over $P_1 + P_2$ can be polylog. (To avoid unduly complicating the statement of our results, we dropped these more exact bounds from the abstract.)

For two integers i and j, we use $[i, j]$ to denote the interval of integers $(i, i+1, \ldots, j-1, j)$. For a matrix A, let $A([i, j], [k, \ell])$ denote the submatrix formed by the rows whose indices are in $[i, j]$ and the columns whose indices are in $[k, \ell]$. Let $A_{i,j}$ denote the i, j entry of A; this use of double subscripts to denote matrix elements will be applied to all matrices considered in the paper. For our notational convenience, the use of single subscripts will be separately and distinctly defined though out this paper for specific matrices (i.e., single subscripts will not necessarily denote a row or column of a matrix).

1.2. Bounded Displacement Rank Matrices

Throughout this paper, we assume indexing of rows and columns of matrices starts at 1.

A matrix A is called *Toeplitz* if the matrix entries on each top-left to bottom-right diagonal are identical, that is

$$\begin{aligned} A_{i,j} &= A_{(i-j)+1,1} \quad \text{if } i \geq j, \\ A_{i,j} &= A_{1,(j-i)+1} \quad \text{otherwise.} \end{aligned}$$

The transpose A^T of a Toeplitz matrix A is Toeplitz, and every block (i.e., contiguous submatrix) of a Toeplitz matrix is Toeplitz. We will compactly represent an $n \times n$ Toeplitz matrix in $2n$ space by specifying the first and last rows.

Kailath and his collaborators ([21, 22]) generalized Toeplitz and Hankel matrices by defining various classes of matrices with bounded *displacement rank*. These matrices can be stored compactly by representing them by their *displacement generators*. We say an $n \times n$ matrix has *displacement rank* δ if it can be represented as a sum of δ products of lower triangular Toeplitz matrices L_i and upper triangular Toeplitz matrices U_i, that is as $\sum_{i=1}^{\delta} L_i U_i$. A displacement rank δ matrix can be compactly represented by specifying the last row of each of the δ triangular Toeplitz matrices $L_1, \ldots, L_\delta$ and the first row of each of the δ triangular Toeplitz matrices $U_1, \ldots, U_\delta$; we will call these 2δ vectors the *displacement generator* of the displacement rank δ matrix (this is sometimes also known as the *compact representation* of a bounded displacement rank matrix).

It is very easy, by use of polynomial product, (see [4]) to transform a bounded displacement rank matrix $A = \sum_{i=1}^{\delta} L_i U_i$ into a sum of $\delta' = \delta + O(1)$ products of upper triangular Toeplitz matrices U'_i and lower triangular Toeplitz matrices U'_i, that is to rewrite A as $\sum_{i=1}^{\delta'} U'_i L'_i$.

Note that any matrix has displacement rank at most n, and a Toeplitz matrix has displacement rank at most two. Throughout this section, we assume that for every input matrix with displacement rank at most δ, a displacement generator is also available, so an $n \times n$ matrix with displacement rank at most δ will be stored in space $2n\delta$.

As an additional benefit, the use of displacement generators gave fast algorithms (see [4, 6, 31, 31, 3, 8, 20]) for basic operations on structured matrices.

LEMMA 1.1. *(see [4, 6, 32]) (a) The product of an $n \times n$ Toeplitz matrix with an n-vector is computable in parallel time $O(\log n)$ with $P(n)$ processors.*
(b) The product of two $n \times n$ Toeplitz matrices is computable (i.e., represented as a sum of products of upper and lower triangular (compactly represented) Toeplitz matrices) in parallel time $O(\log n)$ with $P(n)$ processors.
(c) The product of an $n \times n$ Toeplitz matrix with an $n \times n$ (arbitrary) matrix is computable in parallel time $O(\log n)$ with $nP(n)$ processors.

Proof: see [4, 6, 32].

The next lemma summarizes results on matrices with displacement rank δ that we will be using.

LEMMA 1.2.

(a)[4, 6] The product of an $n \times n$ displacement rank δ matrix and an $n \times n$ displacement rank δ' matrix is computable in parallel time $O(\log n)$ using $\delta\delta' P(n)$ processors. The product matrix has displacement rank at most $\delta\delta' + O(1)$.

(b)[33, 32] Testing nonsingularity of an $n \times n$ Toeplitz matrix and, if so, computing its inverse can be done in parallel time $O(\log^2 n)$ using $nP(n)$ processors. More generally, these tasks can be done for an $n \times n$ displacement rank δ matrix in parallel time $O(\log^2 n)$ using $\delta^2 nP(n)$ processors.

1.3. Motivation

One of the main motivations in this paper is to improve on parallel algorithms for basic linear algebra and combinatorial problems by making them *output sensitive*, that is, the complexity of the algorithm is stated as a function of the output as well as input size. For example, an output sensitive algorithm for computing the rank R of an $n \times n$ matrix may be stated as a function of the output R as well as n. Our output sensitive algorithms for rank have the property that on matrices with sufficiently low rank, they are significantly more efficient than previous algorithms

Recent research has shown a surprisingly deep relation between the complexity of solving important combinatorial problems and the complexity of basic linear algebra problems, such as computing the rank of a matrix. Indeed, as a consequence of this relation, combinatorial problems that are not known to be in $\mathcal{NC}$, such as finding a maximum matching in a graph, have been solved by $\mathcal{RNC}$ algorithms [26]. Our output sensitive parallel algorithms for problems in linear algebra, such as finding the rank of a matrix, will imply output sensitive combinatorial problems such as bipartite matching.

1.4. Our Results

All our algorithms fail with very low likelood $\leq 1/n^{\Omega(1)}$. However all our algorithms are Monte Carlo algorithms: they when they fail they may provide no indication of failure. In contrast, the previous algorithms were Las Vegas algorithms: when they fail, but they provide an indication of failure.

In Section 2, we present an output sensitive randomized algorithm for computing the rank of a matrix. This is a basic algorithmic problem in linear algebra, and extensive research has been conducted on parallel algorithms for it. $\mathcal{NC}$ and $\mathcal{RNC}$ algorithms are now available [5, 29]. Given an $n \times n$ input matrix, the most efficient $\mathcal{RNC}$ rank algorithms run in $T(n)$ time with $M(n)$ processors [25, 24]. Let R denote the rank of the matrix. Our $\mathcal{RNC}$ algorithm runs in parallel time $O(\log n + T(R) \log R)$ using $\pi_{\log n, T(R) \log R}(nP(n), M(R))$ (which is at most $nP(n) + M(R))$ processors.

In Section 3, we study the following classical problem: given an n-set of n-dimensional vectors, find a maximum linearly independent subset (MLIS), that is, a maximum size subset of the input vectors that is linearly independent. This problem differs from the *apparently easier* problem of finding a basis for the space spanned by the input vectors, because a solution for the basis problem may contain arbitrary vectors from the space spanned by the input vectors, whereas a solution to the MLIS problem cannot have vectors other than the input vectors. The importance of the MLIS problem comes from its combinatorial applications (see below). The previous best $\mathcal{RNC}$ algorithms for the MLIS problem are the $T(n)$-time and $nM(n)$-processor algorithm of [5, 23] and the $O(T(n) \log^2 n)$-time and $M(n)$-processor algorithm of [13]. Our new algorithm for MLIS builds on the earlier work in [13], and runs in randomized parallel time $O(T(n) \log n)$ using $M(n)$ processors, but is not output sensitive.

In Section 4, we present an output sensitive randomized algorithm for finding a MLIS of rows in an $n \times n$ matrix of rank R that run either in parallel time $O(\log n + T(R))$ using $\pi_{\log n, T(R)}(nP(n), nM(R))$ (which is at most $n(P(n) + M(R)))$ processors, or in parallel time $O(T(R) \log n)$ using $nP(n)/T(R) + R\, M(R)$ processors, assuming the rank is known (If the rank is not known, then the time bounds of these algorithms only increase by an additive factor of at most $O(T(R) \log R)$, without increase in the processor bounds.).

The improved efficiency of our output sensitive randomized algorithms for finding a MLIS yields improved parallel algorithms for a number of other significant algebraic and combinatorial problems. An immediate application is to the maximum matching problem for an n-vertex graph with a small matching of size R. We show that the maximum matching can be computed in the same parallel bounds as our MLIS algorithm for an $n \times n$ matrix of rank R.

Another application is to the problem of computing matchings and flows in an n-vertex bipartite graph $G = (U, V, E)$ with vertex partition U, V. The case when one vertex subset, say U, is substantially smaller than the other has been studied by [17] and by [2]. This case arises in a number of applications. For example, [17] mentioned the following applications and gave improved algorithms, where the improvements are solely based on the fact that these problems may be modeled by bipartite graphs having one vertex subset of the bipartite partition substantially smaller than the other vertex subset: multiprocessor scheduling with release times and deadlines, a subclass of 0–1 integer programming problems called provisioning or shared fixed cost problems, and the problems of maximum subgraph density, and weighted subgraph density. Let $n_1 = |U|$, $n_2 = |V|$, where $n_1 \ll n_2$. Our Corollary 4.2 shows that a maximum matching can be computed here in randomized parallel time $O(\log n + T(n_1)\log n_1)$ using $\pi_{\log n, T(n_1)}(nP(n), nM(n_1))$ (which is at most $n(P(n) + M(n_1)))$ processors, or in randomized parallel time $O(T(n_1)\log n)$ using $nP(n)/T(n_1) + n_1\ M(n_1)$ processors. This is much improved from the most processor efficient $\mathcal{RNC}$ algorithm, namely, the $O(T(n)\log n)$-time and $nM(n)$-processor algorithm in [16].

Yet other applications are to an output sensitive $\mathcal{RNC}$ algorithm for computing maximum vertex *weighted* matchings (where the weights may be encoded in binary), see [26], and to an output sensitive $\mathcal{RNC}$ algorithm for the two-processor scheduling problem, see [39].

We also consider a frequently occurring class of matrices, namely those with small displacement rank, and gives substantially more efficient randomized parallel algorithms for the rank and MLIS problems on these matrices.

2. OUTPUT SENSITIVE $\mathcal{RNC}$ ALGORITHMS FOR RANK

In the rest of the paper, we use L and U to denote a unit lower triangular $n \times n$ Toeplitz matrix, and a random unit upper triangular $n \times n$ Toeplitz matrix, respectively. In particular, $L_{2,1}, L_{3,1}, \ldots, L_{n,1}$ are $(n-1)$ random numbers chosen independently from any fixed subset of size $n^{O(1)}$ of the field, and L is the matrix with

$$\begin{array}{rcll} L_{i,j} & = & L_{1+(i-j),1} & \text{if } i > j \\ L_{i,j} & = & 1 & \text{if } i = j \\ L_{i,j} & = & 0 & \text{otherwise,} \end{array}$$

$U_{1,2}, U_{1,3}, \ldots, U_{1,n}$ are $(n-1)$ random random numbers chosen independently from a field, and U is the matrix with

$$\begin{array}{rcll} U_{i,j} & = & U_{1,(j-i)+1} & \text{if } j > i \\ U_{i,j} & = & 1 & \text{if } i = j \\ U_{i,j} & = & 0 & \text{otherwise.} \end{array}$$

We consider the following $\mathcal{RNC}$ algorithm of [25] for computing the rank of an $n \times n$ matrix A: Let L and U be as above. Compute the product $H = AL$ followed by the product $G = UH$ in time $O(\log n)$ with $nP(n)$ processors. Let G_R be the leading principal $R \times R$ minor of G, i.e., the submatrix of G indexed by rows $1, \ldots, R$

and by columns $1, \ldots, R$. If $\det G_R \neq 0$, then clearly rank $(A) \geq R$. If $\det G_R = 0$, then with probability $\geq 1 - 1/n^{\Omega(1)}$, A has rank less than R. This follows from the proof of Theorem 2 of [25], noting that the random elements of L, U are chosen over over a fixed set of polynomial size, so the Schwartz-Zippel Lemma [40, 38] insures a failure probability $\leq 1/n^{\Omega(1)}$.

To find the rank, a binary search over $i = 1, \ldots, n$ is executed to find the maximum i such that $\det G_i$ is nonzero (The binary search can be replaced by quicker methods, see [25, 24], but these methods do not seem to yield output sensitive parallel algorithms).

To obtain an output sensitive $\mathcal{RNC}$ algorithm for rank (i.e., one whose processor requirements decrease significantly when the matrix has sufficiently low rank), we replace the binary search by an "exponential search" with $i = 2^0, 2^1, 2^2, 2^3, \ldots$, until we find an ℓ with $\det G_{2^\ell} = 0$. If we do not find such an ℓ, then the matrix A is nonsingular, so the rank is n. If we do find such an ℓ, then with probability $\geq 1/c$ the matrix A is singular, and has rank $< \ell$. Then we do a binary search to find the rank R in the interval $2^{\ell-1}$ to 2^ℓ. Each test for zero determinant on an $R \times R$ matrix is done in $T(R)$ time with $M(R)$ processors. For an input matrix with rank R, ℓ equals $\lceil \log(R+1) \rceil$, so the parallel time for the two searches is $O(\log R)$ times the time for determinant computation. Hence, with constant slowdown, the total parallel time for the determinant computations is $O(T(R) \log R)$ and the processor requirement is $M(R)$. By Proposition 1.1, we have

THEOREM 2.1. *There is an $\mathcal{RNC}$ algorithm for computing the rank R of an $n \times n$ matrix, with probability at least $1 - 1/n^{\Omega(1)}$, in parallel time $O(\log n + T(R) \log R)$ using $\pi_{\log n, T(R) \log R}(nP(n), M(R))$ processors, and $O(n \log^2 n)$ random bits.*

2.1. Application to Matching

Computing the rank of a matrix has many algebraic and combinatorial applications. Consider the problem of finding the cardinality of a maximum matching in a graph (possibly nonbipartite). Let p be a prime $\geq n^2$. We employ the method of [28], and draw m independent random numbers, $x_1, \ldots, x_m$, from the field $\mathbf{Z_p}$, one number per edge. Then we construct the skew symmetric matrix A with

$$\begin{array}{rcll} A_{i,j} & = & x_e & \text{if } e = \{i,j\} \text{ and } i < j \\ A_{i,j} & = & -x_e & \text{if } e = \{i,j\} \text{ and } i > j \\ A_{i,j} & = & 0 & \text{otherwise,} \end{array}$$

and compute its rank. With probability at least $1 - n/p$, the rank equals the cardinality of a maximum matching.

COROLLARY 2.1. *Let G be an n-vertex, m-edge graph, and let R denote the size of a maximum matching in G. R can be computed, with probability $\geq 1 - 1/n^{\Omega(1)}$, by a randomized PRAM in parallel time $O(\log n + T(R) \log R)$ using $\pi_{\log n, T(R) \log R}(nP(n), M(R))$ processors with integer arithmetic operations and $O(m \log n + n \log^2 n)$ random bits.*

2.2. Improvements for Bounded Displacement Rank Matrices

We now assume the input matrix A to our rank algorithm has size $n \times n$ and displacement rank at most δ, so A is given as a sum of δ products of upper and lower triangular (compactly represented) Toeplitz matrices, following our definition of displacement rank given in subsection 1.2. The two $n \times n$ matrices required for our parallel algorithm for rank are AL and $G = UH$. The matrix AL is obtained by multiplying the $n \times n$ matrix A of displacement rank at most δ by an $n \times n$ lower triangular Toeplitz matrix of displacement rank L at most 1. The matrix $G = UH$ is obtained by multiplying an $n \times n$ upper triangular Toeplitz matrix of displacement rank U at most 1 by the $n \times n$ matrix A of displacement rank at most δ.

Recall (see [4]) that polynomial product can be used to transform a bounded displacement rank matrix $A = \sum_{i=1}^{\delta} L_i U_i$ to a sum of $\delta' = \delta + O(1)$ products of upper triangular Toeplitz matrices U'_i and lower triangular Toeplitz matrices U'_i, that is to rewrite A as $\sum_{i=1}^{\delta'} U'_i L'_i$. By Lemma 1.2, these matrix products take parallel time $O(\log n)$ using $\delta P(n)$ processors. The product of two lower (or upper, respectively) triangular Toeplitz matrices is a lower (or upper, respectively) is a triangular Toeplitz matrix. It follows (following our definition of displacement rank given in subsection 1.2), that the matrix $H = AL = (\sum_{i=1}^{\delta'} U'_i L'_i)L$ has displacement rank at most $\delta + O(1)$, since the lower triangular Toeplitz matrix L is multiplied with A from the right. It also follows that the matrix $G = UH$ $= H(\sum_{i=1}^{\delta'} U'_i L'_i)$ has displacement rank at most δ, since the lower triangular Toeplitz matrix U is multiplied with A from the left.

Also, each of the $O(\log R)$ stages of binary search requires a test for a nonsingular submatrix which costs parallel time $T(R)$ using $\delta^2 RP(R)$ processors. Thus the total parallel time for computing the rank remains $O(\log n + T(R) \log R)$, but the processor requirements are reduced, from $\pi_{\log n, T(R)\log R}(nP(n), M(R))$ in the general case, to $\pi_{\log n, T(R)\log R}(\delta P(n), \delta^2 RP(R))$ (which is at most $O(\delta P(n) + \delta^2 RP(R))$) in the displacement rank δ case.

This implies:

THEOREM 2.2. *Given an $n \times n$ displacement rank δ matrix along with a displacement generator of rank δ, the rank R is computable in parallel time $O(\log n + T(R) \log R)$ using $\pi_{\log n, T(R)\log R}(\delta P(n), \delta^2 RP(R))$ processors.*

3. AN $\mathcal{RNC}$ ALGORITHM FOR A MAXIMUM LINEARLY INDEPENDENT SUBSET OF VECTORS

In this section we describe an improvement to Eberly's $\mathcal{RNC}$ MLIS algorithm for computing a maximum linearly independent subset of an n-set of n-dimensional vectors [13]. Our algorithm runs in randomized parallel time $O(T(n) \log n)$, compared to $O(T(n) \log^2 n)$ time for the algorithm of [13]; both algorithms use $M(n)$ processors. For notational convenience, we assume that n is an integer power of two, that is $n = 2^j$, for some integer j.

The MLIS algorithm of [13] uses a divide and conquer strategy, which can be described (thanks to Eberly [15] for this succinct description) as follows:

Input An n-set S of n-dimensional vectors.

[1] Partition the input set of n vectors into two sets of vectors S_1, S_2 each of size $n/2$.
[2] Linearly transform the set of vectors S_2 into a set S_2' so that all vectors of S_2 that linearly depend on the set of vectors S_1 are mapped to the zero vector, and all other vectors of S_2 are mapped to nonzero vectors.
[3] Then the algorithm is applied recursively to both sets of vectors S_1, S_2' in parallel.
Output Those vectors of S that are not mapped to the zero vector (which form a maximum linearly independent subset of S).

In step [2], all linear combinations of the vectors in S_1 are annihilated by the linear transformation being applied to S_2. (So, for example, if S_1 consists of two linearly independent vectors, x_1 and x_2, and S_2 consists of two vectors x_3 and $x_3 + x_1$, where x_1, x_2 and x_3 are linearly independent vectors, then the linear transformation being applied to S_2 will map both vectors in S_2 to the same vector.) Computing this linear mapping in step [2] requires computation of the rank of an $n \times n$ matrix, computation of the inverse of an $n \times n$ matrix, finding a median vector in the final maximum linearly independent subset, and partitioning the input set using that vector. Then the algorithm is applied recursively to both halves in parallel, with an $O(\log n)$ depth of the recursion. Each level of the recursion runs for $O(T(n)\log n)$ time, giving an overall time of $O(T(n)\log^2 n)$. For a fixed level of the recursion, say level $k \in [0, O(\log n)]$, let n_i be the number of nonzero vectors in the ith block of the partition, where $i \in [1, 2^k]$. There are two bottlenecks in the computation of [13] at level k in the computation that resulted in a running time of $O(T(n)\log n)$:

(1) computing the rank of a matrix, and
(2) binary searching for the median vector in the final maximum linearly independent subset.

The first bottleneck does not affect our algorithm since we use the $T(n)$-time $M(n)$-processor rank algorithm of [25, 24]. We circumvent the second bottleneck by *not* computing the median vector. Instead, we simply partition the input set (at the (top) level 0 of recursion) into two blocks by putting the first $n/2$ input vectors in the first half, and putting the last $n/2$ input vectors in the other half. Again, for each $i \in [1, 2^k]$, let n_i be the number of nonzero vectors in the ith block of the partition at level k in the computation and let r_i be the rank of that ith block. Consequently, at level k of our recursion, the resulting 2^k different blocks of the partition of the input vectors may have widely varying ranks r_i ($i \in [1, 2^k]$), i.e., each r_i may have any value from zero to $n/2^k$. However, we claim that the number of processors required at the kth level of the recursion is still $M(n)$. This follows because the sum of the number of vectors over all blocks of the partitions, is $\sum_{i=1}^{2^k} n_i \le \sum_{i=1}^{2^k} n/2^k \le n$, hence the total processor bound is

$$\sum_{i=1}^{2^k} M(n_i) \le 2^k M(n/2^k) \le M(n).$$

The remaining computations at level k (partitioning the input set using that vector, and inverse of an $n \times n$ matrix) run in time at most $T(n)$ using $\sum_{i=1}^{2^k} O(M(n_i)) \le O(M(n))$ processors. By constant slowdown, we have:

THEOREM 3.1. *Given an n-set of n-dimensional vectors, the lexicographically first maximum linearly independent subset (MLIS) is computable by a randomized parallel algorithm in time $O(T(n)\log n)$ using $M(n)$ processors.*

4. OUTPUT SENSITIVE $\mathcal{RNC}$ ALGORITHMS FOR A MAXIMUM LINEARLY INDEPENDENT SUBSET OF VECTORS

To find a maximum set B of linearly independent rows in the matrix A, the $\mathcal{RNC}$ algorithm of [5] computes the ranks $R^{(k)}$ of the n submatrices $A^{(k)}$ of A, where $A^{(k)}$ consists of the first k rows of A, and the algorithm adds row k to B iff $R^{(k)} > R^{(k-1)}$. Note that B is the lexicographically first maximum independent set of rows.

We improve the processor efficiency of this algorithm and make it output sensitive. Recall that for any matrix A, we let $A_{i,j}$ denote the i,j entry of A. First, we construct the random triangular $n \times n$ Toeplitz matrices L and U, as in Section 2. Let $H = AL$ and let $H^{(k)}$ be obtained from H by replacing rows $(k+1), \ldots, n$ by zero rows. For $k = 1, \ldots, n$, we need to compute $G^{(k)} = UH^{(k)}$. Consider the i,j-th entry of $G = G^{(n)}$; this entry equals the inner product of the i-th row of U and the j-th column of H. Let $V^{i,j}$ denote the n-vector whose ℓ-th entry is $U_{i,\ell}H_{\ell,j}$. Let $S^{i,j}$ denote the n-vector of prefix sums of $V^{i,j}$, that is the k-th entry of $S^{i,j}$ equals the sum of the first k entries of $V^{i,j}$. For each $k = 1, \ldots, n$, notice that the i,j-th entry of $G^{(k)}$ equals $\sum_{\ell=1}^{k} U_{i,\ell}H_{\ell,j}$ = the sum of the first k entries of $V^{i,j}$, which is the k-th entry of $S^{i,j}$, that is:

PROPOSITION 4.1. *For $k = 1, \ldots, n$, for each $1 \le i,j \le n$, the i,j-th entry of $G^{(k)}$ equals the k-th entry of $S^{i,j}$.*

For a fixed pair i,j, we compute $V^{i,j}$ in parallel time $O(\log n)$ using $n/\log n$ processors, and the prefix sum $S^{i,j}$ can also be computed in parallel time $O(\log n)$ using $n/\log n$ processors, by the algorithm of Ladner and Fischer [27] (also see Reif [36] and JáJá [19]).

Assume that the rank R of A is known. (Otherwise, it can be computed using the algorithm of Section 2.) Let $G_R^{(k)}$ be the $R \times R$ principal minor of $G^{(k)}$. Since for each $k = 1, \ldots, n$, all the larger principal minors of $G^{(k)}$ are singular, we need to focus only on computing $G_R^{(k)}$. Hence, we need to compute $V^{i,j}$ and $S^{i,j}$ only for the pairs i,j with $i \in [1,R]$ and $j \in [1,R]$. These $S^{i,j}$ give us all the principal minors $G_R^{(k)}$, for $k = 1, \ldots, n$.

Next, for each $k = 1, \ldots, n$, we compute the rank $R^{(k)}$ of each submatrix $G_R^{(k)}$ in parallel, using the $T(n)$-time and $M(R)$-processor algorithm of [24]. The maximum independent set of rows B will contain row k iff $R^{(k)} > R^{(k-1)}$. A parallel time of $O(T(R))$ is achieved by the above computation with $nM(R)$ processors, by running n copies of the R-rank algorithm in parallel. In addition, we must do the multiplication of A by Toeplitz matrix L, which takes $O(\log n)$ time using $nP(n)$ processors. By Proposition 1.1 and constant slowdown, we have total parallel time $O(\log n + T(R))$ using $\pi_{\log n, T(R)}(nP(n), nM(R))$ processors.

We can obtain a time-processor trade off from the above computation; we can decrease processors while increasing parallel time. For $q = 1, \ldots, R$ in parallel, we find the q-th member of the (lexicographically first) maximum independent set of rows B as follows. We execute a binary search over the submatrices $G_R^{(k)}$, $k = 1, \ldots, n$, looking for the first k such that $G_R^{(k)}$ has rank q. For each value of q, this takes parallel time $O(\log n \log^2 R)$ using $M(R)$ processors, with a total of $RM(R)$ processors for the R values of q, summed over $q = 1, \ldots, R$. Notice that the total number of submatrices $G_R^{(k)}$ that we examine is at most $O(R \log n)$, and we only examine in parallel at most $O(R)$ submatrices $G_R^{(k)}$ at a time. Hence, rather than computing all the submatrices $G_R^{(k)}$ $(k \in [1, n])$ in advance, we compute the required $G_R^{(k)}$ "on the fly" in parallel time $O(\log R)$ using $RP(R)$ processors per submatrix, by utilizing precomputed information.

In the precomputation, the submatrix $H([1, n], [1, R])$ is row-wise partitioned into $\lceil n/R \rceil$ distinct $R \times R$ submatrices H_i, where $H_i = H([(i-1)R+1, iR], [1, R])$. Similarly, we column-wise partition the submatrix $U([1, R], [1, n])$ into $\lceil n/R \rceil$ distinct $R \times R$ submatrices U_i, where $U_i = U([1, R], [(i-1)R+1, iR])$, for $i \in [1, \lceil n/R \rceil]$. Then we compute $\lceil n/R \rceil$ distinct $R \times R$ submatrices E_i by multiplying the $R \times R$ submatrices U_i and H_i, to form the product $E_i = U_i H_i$. Finally, for $i = 1, \ldots, \lceil n/R \rceil$, we compute $\lceil n/R \rceil$ prefix sums $F^{(i)} = \sum_{j=1}^{i} E_j$ of the $R \times R$ submatrices E_j. Since the submatrices U_i in the products $U_i H_i$ are Toeplitz, computing each of the E_i costs parallel time $O(\log R)$ using $RP(R)$ processors, and computing the prefix sums costs parallel time $O(\log n)$ using $nR/\log n$ processors, so the overall precomputation takes at most parallel time $O(\log n)$ using $nR/\log n + (\lceil n/R \rceil) RP(R) \leq O(nP(R))$ processors.

Recall that $H^{(k)}$ is obtained from H by replacing rows $(k+1), \ldots, n$ by zero rows, and that row r in H_i corresponds to row $r + (i-1)R$ in H. Hence to compute $G_R^{(k)}$, for any $k \in [1, n]$, fix $i = \lfloor k/R \rfloor$ and construct the $R \times R$ submatrix $H_i^{(k)}$ from H_i by replacing each row r by a zero row, where $r + (i-1)R \in \{(k+1), \ldots, n\}$. Then we multiply the $R \times R$ triangular Toeplitz submatrix U_i by $H_i^{(k)}$, and add the product to $F^{(i-1)}$. Since

$$G_R^{(k)} = U_i H_i^{(k)} + \sum_{j=1}^{i-1} U_j H_j = U_i H_i^{(k)} + \sum_{j=1}^{\lfloor k/R \rfloor - 1} E_j = U_i H_i^{(k)} + F^{(i-1)}$$

$$= U_{\lfloor k/R \rfloor} H_{\lfloor k/R \rfloor}^{(k)} + F^{(\lfloor k/R \rfloor - 1)}$$

it follows that

PROPOSITION 4.2. *For* $k = 1, \ldots, n$, $G_R^{(k)} = U_{\lfloor k/R \rfloor} H_{\lfloor k/R \rfloor}^{(k)} + F^{(\lfloor k/R \rfloor - 1)}$.

Computing one submatrix $G_R^{(k)}$ takes parallel time $O(\log R)$ using $RP(R)$ processors, and so the total processor bound is $R^2 P(R)$ for all the $O(R)$ submatrices $G_R^{(k)}$ that we examine in parallel. This is dominated by the R rank computations on the $G_R^{(k)}$, which can be executed in parallel, in time $O(T(R))$ using $R\ M(R)$ processors. The multiplication of A by Toeplitz matrix L has cost time $O(\log n)$ using $nP(n)$ processors. By Proposition 1.1 and constant slowdown, we have total parallel time $O(\log n + T(R) \log n) \leq O(T(R) \log n)$ using $\pi_{\log n, T(R) \log n}(nP(n), RM(R)) \leq nP(n)/T(R) + R\ M(R)$ processors.

Summarizing these two results, we have:

Theorem 4.1. *Let A be an $n \times n$ matrix whose rank R is known. (If the rank is not known, then the time bounds increase by an additive factor of at most $O(\log^3 R)$, without increase in the processor bounds.) There are $\mathcal{RNC}$ algorithms for computing the lexicographically first maximum independent set of rows of A, with probability at least $1 - 1/n^{\Omega(1)}$ using $O(n \log^2 n)$ random bits, either*

(a)in parallel time $O(\log n + T(R))$ using $\pi_{\log n, T(R)}(nP(n), nM(R))$ processors, or

(b)in parallel time $O(T(R) \log n)$ using $nP(n)/T(R) + R\ M(R)$ processors.

4.1. Combinatorial Applications

The above algorithm has several combinatorial applications. Consider the problem of finding the vertex set of a maximum matching in a graph, i.e., a largest vertex subset B such that the subgraph induced by B has a perfect matching. We first construct the random skew-symmetric matrix A, as described at the beginning of Section 2. Then the vertex set of a maximum matching corresponds to a maximum linearly independent set of rows of A (see Lovasz's Theorem in [35]) and hence can be found by the above algorithm.

This algorithm is useful for improving the efficiency of $\mathcal{RNC}$ matching algorithms when the size of the maximum matching is substantially smaller than the number of vertices. We first run the above algorithm as a preprocessing step to find the vertex set B. Suppose B is of size $|B| = R$. Then we run the algorithm of [16] (which improves [30]) to find a perfect matching on the subgraph induced by B in parallel time $O(T(R) \log R)$ using $RM(R)$ processors.

Corollary 4.1. *Let G be an n-vertex m-edge graph, and let R denote the size of a maximum matching in G. Then a randomized PRAM (with integer arithmetic operations and $O(m \log n + n \log^2 n)$ random bits) can compute, with probability at least $1 - 1/n^{\Omega(1)}$,*

(i)the vertex set of a maximum matching of G in parallel time $O(\log n + T(R))$ using $\pi_{\log n, T(R)}(nP(n), nM(R))$ processors, or in parallel time $O(T(R) \log n)$ using $nP(n)/T(R) + RM(R)$ processors, and
(ii)a maximum matching in parallel time $O(\log n + T(R) \log R)$ using $\pi_{\log n, T(R)}(nP(n), nM(R))$ processors, or in parallel time $O(T(R) \log n)$ using $nP(n)/T(R) + R\ M(R)$ processors.

Our parallel maximum linearly independent set algorithm also gives improvements for computing maximum matchings in certain classes of bipartite graphs. Consider an n-vertex bipartite graph with vertex partition (U, V) where one vertex set of the partition, say U, is substantially smaller than the other. Let $n_1 = |U| \ll n_2 = |V|$. Observe that the maximum size R of a matching in G is $\leq n_1$. Hence the $(n_1 + n_2) \times (n_1 + n_2)$ adjacency matrix of a bipartite graph may be replaced by its upper right $n_1 \times n_2$ submatrix. Thus we can apply our parallel maximum linearly independent set algorithm to this submatrix, with $R = n_1$, and the maximum matching can be computed using the algorithm of [16]. Note that since the adjacency matrix is replaced by its upper right $n_1 \times n_2$ submatrix, we reduce the processor bound from $nP(n)$ to $n_1 P(n)$ to do the multiplication of A by Toeplitz matrix L, (and the precomputations of our algorithm) in time $O(\log n)$. Thus, with constant slowdown, we can in this case replace $nP(n)$ with $n_1 P(n)$ and also substitute n_1 in place of R in the bounds of Corollary 4.1, yielding:

COROLLARY 4.2. *Let $G = (U, V, E)$ be an n-vertex bipartite graph with $n_1 = |U| \ll n_2 = |V|$. Then a randomized PRAM (with integer arithmetic operations) can compute, with probability at least $1 - 1/n^{\Omega(1)}$,*

(i)the vertex set of the maximum matching in parallel time $O(\log n + T(n_1))$ using $\pi_{\log n, T(n_1)}(nP(n), nM(n_1))$ processors, or in parallel time $O(T(n_1)\log n)$ using $nP(n)/T(n_1) + n_1 M(n_1)$ processors, and

(ii)a maximum matching in parallel time $O(\log n + T(n_1)\log n_1)$ using $\pi_{\log n, T(n_1)}(nP(n), nM(n_1))$ processors, or in parallel time $O(T(n_1)\log n)$ using $nP(n)/T(n_1) + n_1\ M(n_1)$ processors.

4.2. Improvements for matrices with small displacement rank

We now give reduced processor requirements (while keeping the time bounds the same) for finding a maximum linearly independent subset of vectors, in the displacement rank δ case. We assume the input matrix A has displacement rank at most δ, so A is given as a sum of δ products of upper and lower triangular (compactly represented) Toeplitz matrices, following our definition of displacement rank given in subsection 1.2. Recall that by Lemma 1.2, testing nonsingularity of an $R \times R$ displacement rank δ matrix and, if so, computing its inverse can be done in parallel time $O(\log^2 R)$ using $\delta^2 R\ P(R)$ processors.

The submatrices of A with consecutive rows and consecutive columns have displacement rank at most $\delta + O(1)$ (in fact, can be shown to have displacement rank at most $\delta + 4$). All the intermediate matrices constructed in the maximum linearly independent subset algorithm are such submatrices of A, which are in some cases multiplied on either side by a triangular Toeplitz matrices. Thus all the intermediate matrices constructed in the algorithm have displacement rank at most $O(\delta)$. Hence these matrices can be stored compactly in $O(\delta n)$ space, instead of the n^2 space required in the general matrix case. Also, we can reduce the processor bounds for the $R \times R$ matrix products and singularity tests to be executed; these processor bounds reduce, from $M(R)$ in the general matrix case, to $\delta^2 R\ P(R)$ processors for matrices of displacement rank at most δ. We conclude that to find a maximum linearly independent subset of vectors, we can improve on Theorem 4.1 as follows:

1. take time $O(\log n + \log^2 R)$ and achieve a processor bound decrease, from $\pi_{\log n, T(R)}\ (nP(n), nM(R))$ in the general matrix case, to $\pi_{\log n, \log^2 R}\ (\delta P(n), \delta^2 nR\ P(R))$ (which is at most $O(\delta(P(n) + \delta nR\ P(R))))$ in the displacement rank δ case, or
2. take time $O((\log n)\log^2 R)$, and achieve a processor bound decrease, from $nP(n)/\log^2 R + R\ M(R)$ in the general matrix case, to $\delta P(n)/\log^2 R + \delta^2 R^2\ P(R)$ in the displacement rank δ case.

This implies:

THEOREM 4.2. *Given an $n \times n$ displacement rank δ matrix along with a displacement generator of rank δ, a maximum linearly independent subset of the rows is computable in either*

(a)parallel time $O(\log n + \log^2 R)$ using $\pi_{\log n, \log^2 R}(\delta P(n), \delta^2 nR\ P(R))$ processors, or

(b)parallel time $O((\log n) \log^2 R)$ using $\delta(P(n)/\log^2 R + \delta R^2\ P(R))$ processors.

These processor bounds can be further substantially improved by the results of Reif [37] in the case where the entries of the input matrices are rational numbers with a bounded number of bits.

5. ACKNOWLEDGMENTS

The problems and applications considered in this paper were proposed by Joseph Cheriyan and he made invaluable contributions to the presentation of these results. The author wishes to thank Deganit Armon, Shenfeng Chen, Zhiyong Li, and Hongyan Wang, insightful edits and comments on the paper, and Ken Robinson for excellent editorial assistance.

REFERENCES

1. A.V.Aho, J.E.Hopcroft and J.D.Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.
2. R.K.Ahuja, J.B.Orlin, C.Stein and R.E.Tarjan, *Improved algorithms for bipartite network flow problems*, Tech. Report, Sloan School of Management, M. I. T., Cambridge, MA, 1990.
3. G.S.Ammar and W.G.Gragg, *Superfast solution of real positive definite Toeplitz systems*, SIAM J. Matrix Anal. Appl. **9** (1988), 61–76.
4. R.R.Bitmead and D.O.Anderson, *Asymptotically fast solution of Toeplitz and related systems of linear equations*, Linear Algebra and its Applications **34**, (1980), 103–116.
5. A.Borodin, J.von zur Gathen and J.Hopcroft, *Fast parallel matrix and GCD computations*, Information and Control **52** (1982), 241–256.
6. R.P.Brent, F.G.Gustavson, and D.Y.Y.Yun, *Fast solution of Toeplitz systems of equations and computation of Padé approximants*, J. Algorithms **1** (1980), 259–295.
7. W.S.Brown and J.F.Traub, *On Euclid's Algorithm and the Theory of Subresultants*, J. ACM **18** (1971), 505–514.
8. J.Chun and T.Kailath, *Divide-and-conquer solution of least-squares problems for matrices with displacement structure*, SIAM J. Matrix Anal. Appl. (submitted).
9. D. G. Cantor and E. Kaltofen, *On fast multiplication of polynomials over arbitrary rings*, Acta Inf. **28** (1991), 697–701.
10. J.Cheriyan, *Random weighted Laplacians, Lovász minimum digraphs and finding minimum separators*, Proc. 4th ACM-SIAM Symposium on Discrete Algorithms (1993), 31–40.
11. J.Cheriyan, *Finding minimum digraph X-Y separators and minimum bipartite vertex covers in randomized $O(M(n))$ time*, Preprint, Department of Combinatorics & Optimization, University of Waterloo, February 1993.
12. D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetic progressions*, J. Symbolic Comp., **9** (1990), 23–52.
13. W. Eberly, *Efficient parallel independent subsets and matrix factorizations*, Proc. 3rd IEEE Symposium on Parallel and Distributed Processing, (1991), 204–211.

14. W. Eberly, *On efficient band matrix arithmetic*, Proc. 33rd Annual IEEE Symposium on F.O.C.S., (1992), 457–463.
15. W. Eberly, personal communication, (1999).
16. Z. Galil and V. Pan, *Improved processor bounds for combinatorial problems in RNC*, Combinatorica **8** (1988), 189–200.
17. D.Gusfield, C.Martel, D.Fernandez-Baca, *Fast algorithms for bipartite network flow*, SIAM J. Computing **16** (1987), 237–251.
18. A. W. Ingleton and M. J. Piff, *Gammoids and transversal matroids*, J. Combinatorial Theory (B) **15** (1973), 51–68.
19. J. JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley (1992).
20. T.Kailath and J.Chun, *Generalized Gohberg-Semencul formulas for matrix inversion*, Op. Theory: Advan. Appl. **40** (1989), 231–246.
21. T.Kailath, S.-Y.Kung, and M.Morf, *Displacement ranks of matrices and linear equations*, J. Math. Anal. Appl. (1979), 395–407.
22. T.Kailath, A.Viera, and M.Morf, *Inverses of Toeplitz operators, innovations, and orthogonal polynomials*, SIAM Rev. **20** (1978), 106–119.
23. E.Kaltofen and V.Pan, *Processor efficient parallel solution of linear systems over an abstract field*, Proc. 3rd Annual ACM Symposium on Parallel Algorithms and Architectures, (1991), 180-191.
24. E.Kaltofen and V.Pan, *Processor-efficient parallel solution of linear systems II The positive characteristic and singular cases*, Proc. 33rd Annual IEEE Symposium on F.O.C.S. (1992), 714–723.
25. E.Kaltofen and B.D.Saunders, *On Wiedemann's method of solving sparse linear systems*, in Proc. AAECC-5, Lecture Notes in Computer Science **536**, 216–226, Springer-Verlag, 1991.
26. R.M.Karp, E.Upfal and A.Wigderson, *Computing a perfect matching is in random NC*, Combinatorica **6** (1986), 35–48.
27. Ladner, R.E., and M.J. Fischer, *Parallel prefix Computation*, JACM **27(4)** (1980), 831–838.
28. L.Lovász, *On determinants, matchings and random algorithms*, Fundamentals of Computing Theory, L.Budach, Ed., Akademia Verlag, Berlin, (1979).
29. K.Mulmuley, *A fast parallel algorithm to compute the rank of a matrix over an arbitrary field*, Combinatorica **7** (1987), 101–104.
30. K.Mulmuley, U.V.Vazirani and V.V.Vazirani, *Matching is as easy as matrix inversion*, Combinatorica **7** (1987), 105–113.
31. B.R.Musicus, *Levinson and fast Choleski algorithms for Toeplitz and almost Toeplitz matrices*, Internal Report, Lab of Elec., MIT (1981).
32. V.Pan, *New effective methods for computations with structured matrices*, Tech. Report 88–28, Computer Science Dept., State Univ. of New York at Albany, Albany, NY.
33. V.Pan and J.H.Reif, *Some polynomial and Toeplitz matrix computations*, Proc. 28th Annual IEEE Symposium on F.O.C.S., (1987), 173–184.
34. A.Pothen, H.D.Simon and K.Liou, *Partitioning sparse matrices with eigenvectors of graphs*, SIAM J. Matrix Anal. Appl. **11** (1990), 430–452.
35. M.O.Rabin and V.V.Vazirani, *Maximum Matchings in General Graphs through Randomization*, J. Algorithms **10** (1989), 557-567.
    Reif [36]
36. J.H.Reif, ed., *Synthesis of Parallel Algorithms*, Morgan Kaufmann (1993).
37. J.H. Reif, *Efficient Parallel Factorization and Solution of Structured and Unstructured Linear Systems,* to appear in Journal of Computer and System Sciences, 2000.
38. J.T.Schwartz, *Fast probabilistic algorithms for verification of polynomial identities*, J. ACM **27** (1980), 701–717.
39. U.V.Vazirani and V.V.Vazirani, *The two-processor scheduling problem is in R-NC*, Proc. 17th Annual ACM S.T.O.C., (1985), 11–21.
40. R. E. Zippel, *Probabilistic algorithms for sparse polynomials*, in "Proc. EUROSAM '79", Lecture Notes in Computer Science **72** (1991), 216–226.