Parallel Derandomization Techniques

Yijie Han

Department of Computer Science
University of Kentucky
Lexington, KY 40506, USA

1 Introduction

With the advance of computer technology it is now possible to build parallel computers with thousands even millions of processors. These parallel computers are capable of executing many computer instructions simultaneously. Thus the study of parallelism of problems to be solved by parallel computers and the design of parallel algorithms become an important research topic. Problems for which trivial sequential algorithms exist are sometimes intriguing problems to be solved in parallel. Many techniques have been invented to exploit parallelism. In this chapter we will outline a powerful technique for the design of parallel algorithms, the derandomization technique.

The basic idea of derandomization is to start with a randomized algorithm and then obtain a deterministic algorithm by applying the technique of derandomization.

Derandomization is a powerful technique because with the aid of randomization the design of algorithms, especially the design of parallel algorithms, for many difficult problems becomes manageable. The technique of derandomization offers us the chance of obtaining a deterministic algorithm which would be difficult to obtain otherwise. For some problems the derandomization technique enables us to obtain better algorithms than those obtained through other techniques.

Although the derandomization technique has been applied to the design of sequential algorithms[Rag] [Sp], these applications are sequential in nature and cannot be used directly to derandomize parallel algorithms. To apply derandomization techniques to the design of parallel algorithms we have to study how to preserve or exploit parallelism in the process of derandomization. Thus we put emphasis on derandomization techniques which allow us to obtain fast and efficient parallel algorithms.

Every technique has its limit, so does the derandomization technique. In order to apply derandomization techniques, a randomized algorithm must be first designed or be available. Although every randomized algorithm with a finite sample space can be derandomized, it does not imply that the derandomization approach is always the right approach to take. Other algorithm design techniques might yield much better algorithms than those obtained through derandomization. Thus it is important to classify situations where derandomization techniques have a large potential to succeed. In the design of parallel algorithms we need identify situations where derandomization techniques could yield good parallel algorithms.

Since derandomization techniques are applied to randomized algorithms to yield deterministic algorithms, the deterministic algorithms are usually derived at the expense of a loss of efficiency (time and processor complexity) from the original randomized algorithms. Thus we have to study how to obtain randomized algorithms that are easy to derandomize and have small time and processor complexities.

2 The Scheme of Derandomization

The basic idea of derandomization is to obtain a deterministic algorithm by first designing a randomized algorithm and then removing the randomness from the randomized algorithm. Derandomization is a powerful tool in the design of algorithms because it is usually easier to design a randomized algorithm than to design a deterministic algorithm. If we know how to design a randomized algorithm and how to derandomize the algorithm we will arrive at a deterministic algorithm.

A straightforward way of derandomization is to examine all the sample points to find a good point. Suppose we know that the expectation of a random variable r is good through a probabilistic analysis. Then there is a sample point p on which the value of r is good. Here r could be a random variable indicating the failure rate, the size of an independent set, the execution time of a procedure, etc.. Good is interpreted as the value is greater than or equal to (less than or equal to) a desired quantity. A sample point on which the value of r is good is called a good point. Since there exists a good point, by examining all sample points we are guaranteed to find a good sample point p. Suppose that a randomized algorithm outputs a value of r, then by using point p we have a deterministic algorithm which outputs a value no worse than $E[r]$.

The last paragraph gives the basic idea of derandomization. This idea, when applied, often yields deterministic algorithms with an exponential number of operations. That is, in the case of a sequential algorithm it gives an exponential time algorithm, and in the case of a parallel algorithm it requires an exponential number of processors to achieve polylogarithmic time. This is caused by the size of the sample space which, in many situations, has an exponential number of points. Two approaches are usually assumed to obtain efficient deterministic algorithms. One is to use a small sample space, the other is to avoid exhaustive search of the sample space. The first approach requires a good design of the sample space while the second approach requires an efficient search strategy.

Both small sample space and fast search techniques are usually required when the derandomization method is used to obtain efficient NC[Co1] [Co2] algorithms. This is because in many situations the search strategy can examine only a constant number of subspaces in polylogarithmic time. There are exceptions where exponential sized sample space is used to achieve efficient parallel algorithms.

Thus to obtain an efficient algorithm through derandomization we have to design a sample space easy to search, to perform a probabilistic analysis showing that the expectation of a desired random variable is no less than demanded and to provide an efficient search technique which ultimately returns a good sample point.

2.1 The Design of Sample Space

Two considerations are usually given to the design of a sample space. One is the size of the sample space, *i.e.* the number of sample points in the space. The other is the independence of the random variables in the sample space.

It usually takes less resource to search a small sample space than it is to search a large one. This is particularly so when exhaustive search technique is used to locate a good sample point. In the case of the design of a parallel algorithm, a smaller sample space implies that fewer processors are needed in order to examine all the points in the sample space.

The independence of random variables in the sample space is also an important factor. When binary search technique is used to search the sample space, the random variables can be fixed one by one. In this case we usually want the random variables to be fixed independent of other unfixed random variables. The independence condition usually helps the probabilistic analysis showing that after fixing the random variables the remaining smaller sample space contains a good point. In many situations the use of independence also helps the computation of the expectation of the subspace and thus facilitates the design of an efficient algorithm. In the design of parallel algorithms the independence condition sometimes helps to fix several random variables simultaneously and independently, thus speeding up the derandomization process. In these situations we usually require large degree independence among random variables.

Mutually independent random variables are used in the design of sequential algorithms[Rag] [Sp]. When n 0/1-valued uniformly distributed mutually independent random variables are needed in the design of a randomized algorithm, the sample space $S = \{0, 1\}^n$ can be used. Such a sample space contains an exponential number of sample points. Efficient search technique is needed to locate a good point.

In the design of parallel algorithms we usually want a small sample space. This can be achieved by using limited independence among random variables. When n 0/1-valued uniformly distributed pairwise independent random variables are needed, a sample space containing $O(n)$ points can be used. Let $k = \lceil \log n \rceil$. The sample space is $\Omega = \{0,\ 1\}^{k+1}$. For each $a = a_0 a_1 ... a_k \in \Omega$, $Pr(a) = 2^{-(k+1)}$. The value of random variables x_i, $0 \leq i < n$, on point a is $x_i(a) = (\sum_{j=0}^{k-1}(i_j \cdot a_j) + a_k) \bmod 2$, where i_j is the j-th bit of the binary expansion of i. This design is given by Luby[L2].

To obtain n 0/1-valued uniformly distributed k-wise independent random variables, a set of n vectors $S = \{i | i \in Z_2^l\}$ in which every k vectors are linearly independent can be used, where l is a function of n and k. The random variable x_i corresponding to $i \in S$ has value $x_i(a) = \sum_{j=0}^{l-1}(i_j \cdot a_j) \bmod 2$. It can be verified that x_i's are k-wise independent[BR]. When NC algorithms are to be designed using the binary search technique to locate a good sample point the sample space can not be larger than $n^{\log^c n}$, therefore we can use n random variables with at most $(\log^c n)$-wise independence[BR], where c is a constant. In this case l is bounded by $O(\log^{c+1} n)$.

The above mentioned designs only give random variables distributed uniformly on $\{0, 1\}$. When random variables with range greater than 2 or random variables distributed nonuniformly on $\{0, 1\}$ are needed, the sample space can be designed as follows. Let $q \geq n$ be a prime. n random variables which are d-wise independent and uniformly distributed on $\{0, 1, ..., q-1\}$ can be designed on a sample space S containing q^d points. Each point in S is assigned probability $1/q^d$. Let $x =< x_0, x_1, ..., x_{d-1} >$ be a point in S. The value of random variable r_i on x is $r_i(x) = (\sum_{j=0}^{d-1} x_j \cdot i^j) \bmod q$. This design is given by Joffe[Jo] and by Luby[L1]. By imposing a function $f : \{0, 1, ..., q-1\} \mapsto \{0, 1, ..., p\}$ on random variable r_i we obtain a random variable taken value from set $\{0, 1, ..., p\}$. Luby[L1] used a table to implement function f and therefore obtained random variables nonuniformly distributed on $\{0, 1\}$. Although the designed sample space is rather compact in that it contains only q^d points, it is not easy to evaluate conditional expectation on a subspace and not easy to search the sample space using binary search method. Thus this space is a good design when exhaustive search method is used to convert a randomized algorithm to a deterministic algorithm. In particular, Luby used exhaustive search on such a sample space to obtain a fast parallel algorithm for the maximal independent set problem[L1].

Another design given by Luby[L2] is to use a set of 0/1-valued uniformly distributed pairwise independent random variables to construct random variables which are not uniformly distributed. Let $R_i = \{r_{i0}, r_{i1}, ..., r_{i,n-1}\}$ be a set of 0/1-valued uniformly distributed pairwise independent random variables. Take k sets, $R_0, R_1, ...R_{k-1}$. Let the random variables in R_i's be mutually independent. Then the random variables $x_j = \sum_{t=0}^{k-1} r_{tj} \cdot 2^t$, $0 \leq j < n$, are pairwise independent random variables uniformly distributed on $\{0, 1, ..., 2^k - 1\}$. A function can then be imposed on this range to obtain nonuniformly distributed random variables. This sample space contains $O(n^k)$ points. When k is not a constant the sample space contains more than a polynomial number of points. Thus exhaustive search can not be used on this sample space when k is not a constant. However, this sample space is well suited for binary search. Luby showed how to search this sample space to obtain efficient parallel algorithms for the $\Delta + 1$ vertex coloring problem, the maximal independent set problem and the maximal matching problem[L3].

Recently Han and Igarashi gave a new design of a sample space on which n 0/1-valued uniformly distributed mutually independent random variables can be built[HI]. n 0/1-valued uniformly distributed mutually independent random variables r_i, $0 \leq i < n$, are used. Without loss of generality assume that n is a power of 2. A tree T which is a complete binary tree with n leaves plus a node which is the parent of the root of the complete binary tree (thus there are n interior nodes in T and the root of T has only one child). n variables $x_0, x_1, ..., x_{n-1}$ are associated with n leaves of T and the n random variable r_i's are associated with the interior nodes of T. The n leaves of T are numbered from 0 to $n-1$. Variable x_i is associated with leaf i. Variables x_i, $0 \leq i < n$, are randomized as follows. Let r_{i_0}, r_{i_1}, ..., r_{i_k} be the random variables on the path from leaf i to the root of T, where $k = \log n$. Random variable x_i is defined to be $x_i = (\sum_{j=0}^{k-1} i_j \cdot r_{i_j} + r_{i_k}) \bmod 2$. It can be verified that random variables x_i, $0 \leq i < n$, are uniformly distributed mutually independent random variables. Tree T is called a random variable tree. This design is particularly suited to the derandomization of certain random variables[HI].

Combine Luby's design[L2] and Han and Igarashi's design[HI] we can obtain mutually independent random variables uniformly distributed in $\{0, 1, ..., 2^k - 1\}$ as follows. Let $R_i = \{r_{i0}, r_{i1}, ..., r_{i,n-1}\}$ be a set of 0/1-valued uniformly distributed mutually independent random variables. Take k sets, $R_0, R_1, ...R_{k-1}$. Let the random variables in R_i's be mutually independent. Then the random variables $x_j = \sum_{t=0}^{k-1} r_{tj} \cdot 2^t$, $0 \leq j < n$, are mutually independent random variables uniformly distributed on $\{0, 1, ..., 2^k - 1\}$. We can then apply a function on this range to obtain nonuniformly distributed random variables.

It is far from clear what kind of sample space is the best for the design of parallel algorithms through derandomization. The choice of the sample space is related to the probabilistic analysis, the search technique, and therefore the problem to be solved. The works by Berger and Rompel[BR], Luby[L2], Han and

Igarashi[HI] [H3] [H4] began to demonstrate certain design paradigms for the construction of sample spaces good for achieving efficient parallel algorithms.

2.2 The Probabilistic Analysis

The probabilistic analysis is needed first to show that the expectation of a desired random variable is no worse than demanded. Since the expectation of the random variable is good, there must exists a good point. In the case when the space partition method is used to locate a good sample point, conditional expectation has to be analyzed. Luby's analysis for the vertex partition problem[L2] provides an excellent example here to showcase the probabilistic analysis of a problem.

The vertex partitioning problem is to label vertices of a graph $G = (V, E)$ with 0's and 1's, $l : V \mapsto \{0, 1\}$, such that the size of the crossing set $|\{(i, j)|(i, j) \in E, l(i) \neq l(j)\}| \geq |E|/2$. $|V|$ 0/1-valued uniformly distributed mutually independent random variables x_i, $0 \leq i < n$, can be used[L2], one for each vertex. For $(i, j) \in E$, $f(x_i, x_j) = x_i \oplus x_j$ is 1 iff (i, j) is in the crossing set, where x_i (x_j) is the random variable associated with vertex i (j) and $\oplus$ is the exclusive-or operation. The total number of edges in the crossing set can be expressed as $F = \sum_{(i,j)\in E} f(x_i, x_j)$. The expectation of F is $E[F] = |E|/2$. Thus there exists a sample point p such that $F(p) \geq |E|/2$.

Since n mutually independent random variables are used, the sample space contains an exponential number of points. The sample space designed by Han and Igarashi[HI] can be used to obtain n mutually independent random variable. When a random variable tree is used, we can view the random variable x_i at the leaf of the random variable tree as a function of random variable r_i's in the interior nodes of the tree. Thus we transform the original problem of locating a good point $(x_0, x_1, ..., x_{n-1})$ to the problem of locating a good point $(r_0, r_1, ..., r_{n-1})$. The conditional expectation after setting a random variable r at the lowest level in the random variable tree can be easily obtained. Let r be such a random variable and let x_i and $x_{i\#0}$ be the two children of r in the random variable tree, where $i\#0$ is the value obtained by complementing the 0-th bit of i. The conditional expectation when r is set is $E[F|r = 0] = E[F|x_i = x_{i\#0}]$ and $E[F|r = 1] = E[F|x_i = 1 - x_{i\#0}]$. The nice feature of Han and Igarashi's design is that all random variables at the lowest level of the random variable tree can be set simultaneously. Note that there are $n/2$ random variables at that level.

Luby designed a sample space which has only $O(n)$ points[L2]. Each point is assigned equal probability. The value of random variable x_i on point r is $(\sum_{k=0}^{\log n-1} i_k \cdot r_k + r_{\log n})$ mod 2, where i_k (r_k) is the k-th bit of i (r). Thus Luby's design transforms the problem of locating a good point $(x_0, x_1, ..., x_{n-1})$ to the problem of locating a good point $(r_0, r_1, ..., r_{\log n})$. For function $f(x_i, x_j) = x_i \oplus x_j$, let $t = \min\{u|i_v = j_v \text{ for } v \geq u\}$, when r_k, $k < t$, is fixed, $E[f(x_i, x_j)]$ does not change. When r_t is fixed $E[f] = (f(0, 0) + f(1, 1))/2$ or $E[f] = (f(0, 1) + f(1, 0))/2$ depends on whether r_t is set to 0 or 1. Thus the sample space designed by Luby also has the feature that conditional expectations can be evaluated easily.

Probabilistic analysis is essential for establishing a randomized algorithm in the first place. It is also vital to the feasibility of an efficient derandomization process. When space partition methods are used to search for a good point the conditional expectations need to be evaluated. Probabilistic analysis usually provides ways for evaluating these conditional expectations.

2.3 The Search Technique

The known search techniques can be put into three categories, namely exhaustive search, binary search and multiple space partition search.

When the exhaustive search method is used all points in the sample space are examined and the best point is chosen as the output. Exhaustive method can only be used when the size of the sample space is small. In the design of sequential algorithms the size of the sample space can not be larger than a polynomial if polynomial time algorithm is demanded. In the design of parallel algorithms the size of the sample space can not be larger than a polynomial if NC algorithms are demanded. Although exhaustive search method is sometimes considered as a brute force method, in many situations it yields very fast algorithms. In situations where the sample space and the probabilistic analysis are complicated the exhaustive search may be the only feasible method to search the sample space.

Binary search is usually considered the "standard" way to search the sample space efficiently. When the sample space contains n points and binary search is used to partition the space evenly then $\log n$ steps are enough to locate a good point. In contrast, the exhaustive method requires n operations. When function f has expectation $E[f|S_i]$ on space S_i, $i = 1, 2, 3$, and $S_1 = S_2 \cup S_3$, $S_2 \cap S_3 = \phi$, then either $E[f|S_2] \geq E[f|S_1]$ or $E[f|S_3] \geq E[f|S_1]$. Thus if the expectation of f on S_1 is good then the expectation of f is good on at least one of S_2 and S_3. Thus if the expectation of f on S_i, $1 \leq i \leq 3$, can be computed efficiently the search can continue on either S_2 or S_3. Binary search methods have been used by Raghaven[Rag], Spencer[Sp] in obtaining sequential algorithms through derandomization. Luby[L2] and Berger and Rompel[BR] used binary search to obtain efficient parallel algorithms through derandomization.

Multiple space partition method is a method used in the design of parallel algorithms. It partitions the sample space into several subspaces and evaluate the conditional expectation on each of the subspace. A good subspace is chosen for the continuing search until a good point is located. The method is typically used in the design of parallel algorithms to speed up the computation when extra processor power is available to evaluate conditional expectations of subspaces. Luby[L2] used this method to speed up the derandomization process of the PROFIT/COST problem. Not much is known on how to perform multiple space partition without resorting to extra processor power. Han and Igarashi[HI] showed a case where a sample space of size n is partitioned into $\sqrt{n}$ subspaces and a good subspace is found using only a linear number of processors.

Luby[L2] gave a technique for searching a sample space containing nonuniformly distributed 0/1-valued random variables. Let $\vec{y} =< y_i \in \{0,1\}^p : i = 0, 1, ..., n-1 >$. Let $\vec{x_u}$, $p \leq u < q$, be totally independent random bit strings, each of length n. Let $B(\vec{x_{q-1}} \cdots \vec{x_1}\vec{x_0})$ be the function we are to search for a good point. Let $\vec{z}$ be a vector of n bits. Define $TB(\vec{y}) = E[B(\vec{x_{q-1}} \cdots \vec{x_{p+1}}\vec{x_p}\vec{y})]$. Then $E[TB(\vec{x_p}\vec{y})] = E[E[B(\vec{x_{q-1}} \cdots \vec{x_{p+1}}\vec{z}\,\vec{y})| \vec{z} = \vec{x_p}]] = E[B(\vec{x_{q-1}} \cdots \vec{x_p}\vec{y})] = TB(\vec{y})$. Luby's method is to find a $\vec{z}$ satisfying $TB(\vec{z}\,\vec{y}) \geq E[TB(\vec{x_p}\vec{y})] = TB(\vec{y})$, thus fixing the random bits in $\vec{x_p}$. Luby's method[L2] [L3] can be interpreted as follows. Solving the big problem by solving q small problems, one for each $\vec{x_u}$. These small problems are solved sequentially. After the small problems for $\vec{x_u}$, $0 \leq u < v$, are solved. Conditional expectations are evaluated based on the setting of bits x_{i_u}, $0 \leq u < v$, $0 \leq i < n$, and the small problem for fixing $\vec{x_v}$ is ready to be solved.

2.4 Approximation

Approximation method is a method to simulate the derandomization process by substituting the function to be search on with an approximation function. This method is typically used in situations where conditional expectation is hard to evaluate. With the use of an approximation function, the evaluation of conditional expectations becomes the evaluation of function values under certain restrictions. The approximation methods allow us to extend the applicability of derandomization. Raghaven[Rag] used this method in the design of a sequential algorithm. Luby[L2] used this method in the design of a parallel algorithm. The main problem involved here is to find a suitable approximation function. We will not discuss the approximation method in detail because the method is very much case-dependent and it usually involves complicated analysis.

3 Derandomization Algorithms

In this section we will present derandomization algorithms for a class of functions. These functions model several important problems for which fast and efficient deterministic parallel algorithms can be obtained through derandomization. The derandomization algorithms presented in this section are given in [BR] [HI] [H3] [H4] [L2]. The parallel machine model we use is the Parallel Random Access Machine (PRAM)[FW]. In a PRAM the memory cells are shared among processors. Each memory cell can be accessed by any processor in a step. PRAMs are usually classified into the Exclusive Read Exclusive Write(EREW) PRAM in which a memory cell cannot be read or written by more than one processor in a step, the Concurrent Read Exclusive Write(CREW) PRAM in which a memory cell can be read by several processors in a step, but simultaneous write to the same memory cell by several processors is not allowed, the Concurrent Read Concurrent Write(CRCW) PRAM in which a memory cell can be read or written by several processors in a step.

3.1 Pairwise Independent Random Variables

We are to derandomize for functions of the form

$$B(\vec{x}) = \sum_i f_i(x_i) + \sum_{(i,j)} f_{i,j}(x_i, x_j)$$

.

Such functions arises in many cases, Luby formulated[L3] the vertex partitioning problem, the Δ+1 vertex coloring problem, the maximal independent set problem and the maximal matching problem by functions of such form. Functions f_i and $f_{i,j}$ are called PROFIT/COST functions and function B is called a BENEFIT function[L3]. The problem of derandomizing such a function is called the PROFIT/COST problem[L2]. The formal definition is given below.

Let $\vec{x}=< x_i \in \{0,1\}^q : i = 0, ..., n-1 >$. Each point $\vec{x}$ out of the 2^{nq} points is assigned probability $1/2^{nq}$. Given function $B(\vec{x}) = \sum_i f_i(x_i) + \sum_{i,j} f_{i,j}(x_i, x_j)$, where f_i is defined as a function $\{0,1\}^q \rightarrow \mathcal{R}$ and $f_{i,j}$ is defined as a function $\{0,1\}^q \times \{0,1\}^q \rightarrow \mathcal{R}$. The general pairs PROFIT/COST (GPC for short) problem is to find a good point $\overrightarrow{y}$ such that $B(\overrightarrow{y}) \geq E[B(\overrightarrow{x})]$. B is called the general pairs BENEFIT function and $f_{i,j}$'s are called the general pairs PROFIT/COST functions.

A special case of the GPC problem, called the bit pairs PROFIT/COST (BPC) problem, is a GPC problem in which $q = 1$. In a BPC problem function B is called the BPC BENEFIT function and f_i's and $f_{i,j}$'s are called the BPC PROFIT/COST functions.

For the simplicity of discussion we only consider BENEFIT functions of the form $B(\overrightarrow{x}) = \sum_{i,j} f_{i,j}(x_i, x_j)$.

3.1.1 The Bit Pairs PROFIT/COST Problem

For the bit pairs PROFIT/COST problem we obtain n mutually independent random variables using the following sample space[H3] [HI].

n 0/1-valued uniformly distributed mutually independent random variables r_i, $0 \leq i < n$, are used. Without loss of generality assuming n is a power of 2. We build a tree T which is a complete binary tree with n leaves plus a node which is the parent of the root of the complete binary tree (thus there are n interior nodes in T and the root of T has only one child). The n random variables $x_0, x_1, ..., x_{n-1}$ of B are associated with n leaves of T and the n random variables $r_0, r_1, ..., r_{n-1}$ are associated with the interior nodes of T. The n leaves of T are numbered from 0 to $n-1$. Variable x_i is associated with leaf i.

We now randomize the variables $x_i, 0 \leq i < n$. Let $r_{i_0}, r_{i_1}, ..., r_{i_k}$ be the random variables on the path from leaf i to the root of T, where $k = \log n$. Random variable x_i is defined to be $x_i = (\sum_{j=0}^{k-1} i_j \cdot r_{i_j} + r_{i_k}) \bmod 2$.

Lemma 1: Random variables x_i, $0 \leq i < n$, are uniformly distributed mutually independent random variables.

Proof: By flipping the random bit at the root of the random variable tree we see that each x_i is uniformly distributed in $\{0, 1\}$. To show the mutual independence we note that the mapping $m(\overrightarrow{r}) \mapsto \overrightarrow{x}$, where $x_i = (\sum_{j=0}^{\log n-1} i_j \cdot r_{i_j} + r_{i_{\log n}}) \bmod 2$, is a one to one mapping. Thus $Pr(x_{i_1} = a_1, x_{i_2} = a_2, ..., x_{i_k} = a_k) = 2^{n-k}/2^n = 1/2^k = Pr(x_{i_1} = a_1)Pr(x_{i_2} = a_2) \cdots Pr(x_{i_k} = a_k)$. □

Tree T is called a random variable tree.

We are to find a sample point $\overrightarrow{r} = (r_0, r_1, ..., r_{n-1})$ such that $B(\overrightarrow{r}) \geq E[B] = \frac{1}{4} \sum_{i,j}(f_{i,j}(0, 0) + f_{i,j}(0, 1) + f_{i,j}(1, 0) + f_{i,j}(1, 1))$.

We fix random variables r_i (setting their values to 0's and 1's) one level in a step starting from the level next to the leaves (we shall call this level level 0) and going upward on the tree T until level k. Since there are $k + 1$ interior levels in T all random variables will be fixed in $k + 1$ steps.

Now consider fixing random variables at level 0. Since there are only two random variables x_j, $x_{j\#0}$ which are functions of random variable r_i (node r_i is the parent of the nodes x_j and $x_{j\#0}$) and x_j, $x_{j\#0}$ are not related to other random variables at level 0, and since random variables at level 0 are mutually independent, they can be fixed independently.

Consider in detail the fixing of r_i which is only related to x_j and $x_{j\#0}$. We simply compute $f_0 =$

$f_{j,j\#0}(0,\ 0) + f_{j,j\#0}(1,\ 1) + f_{j\#0,j}(0,\ 0) + f_{j\#0,j}(1,\ 1)$ and $f_1 = f_{j,j\#0}(0,\ 1) + f_{j,j\#0}(1,\ 0) + f_{j\#0,j}(0,\ 1) + f_{j\#0,j}(1,\ 0)$. If $f_0 \geq f_1$ then set r_i to 0 else set r_i to 1. This scheme allows all random variables at level 0 be set in parallel in constant time.

If r_i is set to 0 then $x_i = x_{i\#0}$, if r_i is set to 1 then $x_i = 1 - x_{i\#0}$. Therefore after r_i is fixed, x_i and $x_{i\#0}$ can be combined. The n random variables x_i, $0 \leq i < n$, can be reduced to $n/2$ random variables. BPC functions $f_{i,j}, f_{i\#0,j}, f_{i,j\#0}$, and $f_{i\#0,j\#0}$ can also be combined into one function. It can be checked that the combining can be done in constant time using a linear number of processors.

During the combining process variables x_i and $x_{i\#0}$ are combined into a new variable $x^{(1)}_{\lfloor i/2 \rfloor}$, functions $f_{i,j}, f_{i\#0,j}, f_{i,j\#0}$, and $f_{i\#0,j\#0}$ are combined into a new function $f^{(1)}_{\lfloor i/2 \rfloor, \lfloor j/2 \rfloor}$. After combining a new function $B^{(1)}$ is formed which has the same form of B but has only $n/2$ variables. As we stated above, $E[B^{(1)}] \geq E[B]$.

What we have explained above is the first step of the algorithm [HI]. This step takes constant time using a linear number of processors. After k steps the random variables at levels 0 to $k-1$ in the random variable tree are fixed, the n random variables $\{x_0, x_1, ..., x_{n-1}\}$ are reduced to $n/2^k$ random variables $\{x^{(k)}_0, x^{(k)}_1, ..., x^{(k)}_{n/2^k-1}\}$, functions $f_{i,j}$, $i, j \in \{0, 1, ..., n-1\}$, have been combined into $f^{(k)}_{i,j}$, $i, j \in \{0, 1, ..., n/2^k - 1\}$.

After $\log n$ steps $B^{(\log n)} = f^{(\log n)}_{0,0}(x^{(\log n)}_0, x^{(\log n)}_0)$. The bit at the root of the random variable tree is now set to 0 if $f^{(\log n)}_{0,0}(0,0) \geq f^{(\log n)}_{0,0}(1,1)$, and 1 otherwise. Thus Han and Igarashi's algorithm[HI] solves the BPC problem in $O(\log n)$ time with a linear number of processors.

Theorem 2[HI]: A sample point $\vec{r} = (r_0,\ r_1,\ ...,\ r_{n-1})$ satisfying $B|_{\vec{r}} \geq E[B]$ can be found in $O(\log n)$ time using a linear number of processors and $O(n^2)$ space. $\Box$

A close examination of the process of derandomization of the algorithm shows that functions $f_{i,j}$ are combined according to the so-called file-major indexing for the two dimensional array, as shown in Fig. 1. In the file-major indexing the $n \times n$ array A is divided into four subfiles $A_0 = A[0..n/2-1,\ 0..n/2-1]$, $A_1 = A[0..n/2-1,\ n/2..n-1]$, $A_2 = A[n/2..n-1,\ 0..n/2-1]$, $A_3 = A[n/2..n-1,\ n/2..n-1]$. Any element in A_i proceeds any element in A_j if $i < j$. The indexing of the elements in the same subfile is recursively defined in the same way. The indexing of function $f_{i,j}$ is the number at the i-th row and j-th column of the array. After the bits at level 0 are fixed by our algorithm, functions indexed $4k$, $4k+1$, $4k+2$, $4k+3$, $0 \leq k < n^2/4$ will be combined. After the combination of these functions they will be reindexed. The new index k will be assigned to the function combined from the original functions indexed $4k$, $4k+1$, $4k+2$, $4k+3$. This allows the recursion in our algorithm to proceed.

$$\begin{array}{cccc} \mathbf{0} & \mathbf{1} & \mathbf{4} & \mathbf{5} \\ \mathbf{2} & \mathbf{3} & \mathbf{6} & \mathbf{7} \\ \mathbf{8} & \mathbf{9} & \mathbf{12} & \mathbf{13} \\ \mathbf{10} & \mathbf{11} & \mathbf{14} & \mathbf{15} \end{array}$$

Fig. 1. File-major indexing for the two dimensional array

Obviously we want the input to be arranged by the file-major indexing. When the input has been arranged by the file-major indexing, we are able to build a tree which reflects the way input functions $f_{i,j}$'s are combined as the derandomization process proceeds. We shall call this tree the derandomization tree. This tree is built as follows.

We use one processor for each function $f_{i,j}$. These BPC functions are stored in an array. Let f_{i_1,j_1} be the function stored immediately before $f_{i,j}$ and f_{i_2,j_2} be the function stored immediately after $f_{i,j}$. By looking at (i_1, j_1) and (i_2, j_2) the processor could easily figure out at which step of the derandomization $f_{i,j}$ should be combined with f_{i_1,j_1} or f_{i_2,j_2}. This information allows the tree to be built for the derandomization process. This tree has $\log n + 1$ levels. The functions at the 0-th level (leaves) are those to be combined into a new function which will be associated with the parent of these leaves. The combination happens immediately after the random variables at level 0 in the random variable tree are fixed. In general, functions at level i will be combined immediately after the random variables at level i in the random variable tree are fixed. Note that the term level in the derandomization tree corresponds to the level of the random variable tree. Thus, a node at level i of the derandomization tree could be at depth $\leq \log n - i$.

Since the derandomization tree has height at most $\log n$, it can be built in $O(\log n)$ time using m processors. If the input is arranged by the file-major indexing, the tree can be built in $O(\log n)$ time using optimal $m/\log n$ processors by a careful processor scheduling.

A derandomization tree is shown in Fig. 2.

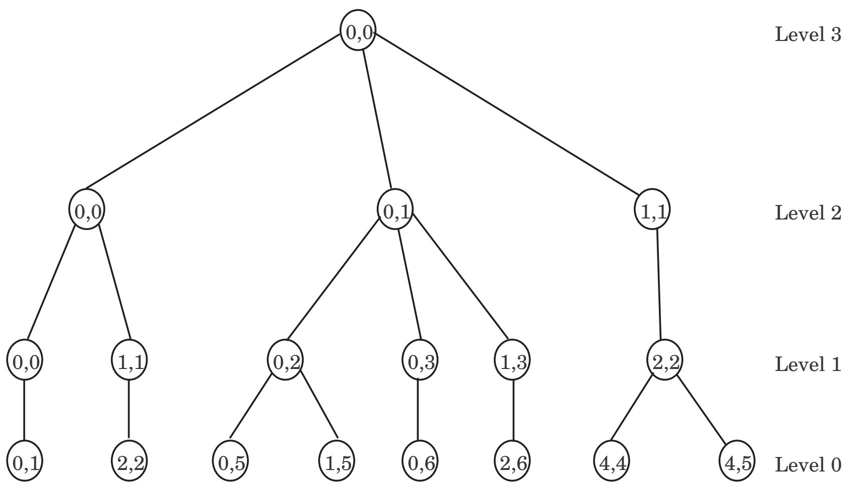

Fig. 2. A derandomization tree. Pairs in the circles are the subscripts of PROFIT/COST functions.

The derandomization process can now be described in terms of the derandomization tree. Combine the functions at level i of the derandomization tree immediately after the random variables at level i of the random variable tree are fixed. The combination can be accomplished by the RAKE[MR] operation. The whole process of the derandomization can now be viewed as a process of tree contraction which uses only the RAKE operation without using the COMPRESS operation[MR].

If the input is not arranged by the file-major indexing we could sort the input into the file-major indexing.

Known sorting algorithms[AKS] [C] have time complexity $O(\log n)$ using a linear number of processors.

Theorem 3[HI]: A sample point $\vec{r} = (r_0, r_1, ..., r_{n-1})$ satisfying $B(\vec{r}) \geq E[B]$ can be found on the CREW PRAM in $O(\log n)$ time using m processors and $O(m)$ space. $\Box$

We are able to obtain an optimal algorithm if the input is arranged by the file major indexing. Since no sorting is needed, the time complexity of the algorithm we described above takes $O(\frac{m}{p} + \frac{n \log n}{p} + \log n)$ time with p processors. The term $n \log n/p$ is incurred due to the fact that we have to keep updated information with which variable x_i, $0 \leq i < n$, has been combined into. This takes $O(n)$ operations in each step of the algorithm.

To obtain an optimal algorithm with time complexity $O(m/p + \log n)$ we first reduce the number of random variables to $n/\log n$. This is done by dividing n random variables into $n/\log n$ groups with random variable x_i, $k \log n \leq i < (k+1)\log n$ in group k. We solve the BPC problem for each group, in parallel for all groups, using a modified version of Luby's algorithm[L2] which can be made to run in $O(\frac{m}{p} + \log\log n)$ time. Upon finishing there are $n/\log n$ random variables left and our original algorithm with time $O(\frac{m}{p} + \frac{n \log n}{p} + \log n)$ now takes $O(\frac{m+n}{p} + \log n)$ time when n is replaced with $n/\log n$.

Theorem 4[HI]: A sample point $\vec{r} = (r_0, r_1, ..., r_{n-1})$ satisfying $B(\vec{r}) \geq E[B]$ can be found on the CREW PRAM in $O(\log n)$ time using optimal $m/\log n$ processors and $O(m)$ space if the input is arranged by the file-major indexing. $\Box$

3.1.2 The General Pairs PROFIT/COST Problem

We shall present a scheme where the GPC problem is solved by pipelining the BPC algorithm to solve BPC problems in the GPC problem.

First we give a sketch of our approach. The incompleteness of the description in this paragraph will be fulfilled later. Let P be the GPC problem we are to solve. P can be decomposed into q BPC problems to be solved sequentially. Let P_u be the u-th BPC problem. Imagine that we are to solve P_u, $0 \leq u < k$, in one pass, *i.e.*, we are to fix $\vec{x_0}$, $\vec{x_1}$, ..., $\vec{x_{k-1}}$ in one pass, with the help of enough processors. For the moment we can have a random variable tree T_u and a derandomization tree D_u for P_u, $0 \leq u < k$. In step j our algorithm will work on fixing the bits at level $j - u$ in T_u, $0 \leq u \leq \min\{k-1, j\}$. The computation in each tree D_u proceeds as we have described in the last subsection. Note that BPC functions $f_{i_v,j_v}(x_{i_v}, x_{j_v})$ depends on the setting of bits x_{i_u}, x_{j_u}, $0 \leq u < v$. The main difficulty with our scheme is that when we are working on fixing $\vec{x_v}$, $\vec{x_u}$, $0 \leq u < v$, have not been fixed yet. The only information we can use when we are fixing the random variables at level l of T_u is that random variables at levels 0 to $l + c - 1$ are fixed in T_{u-c}, $0 \leq c \leq u$. This information can be accumulated in the pipeline of our algorithm and transmitted on the *bit pipeline trees.* Fortunately this information is sufficient for us to speed up the derandomization process without resorting to too many processors. For the sake of a clear exposition we first describe a CREW derandomization algorithm. We then show how to convert the CREW algorithm to an EREW algorithm.

Suppose we have $c\sum_{i=0}^{k}(m * 4^i)$ processors available, where c is a constant. Assign $cm * 4^u$ processors to work on P_u for $\vec{x_u}$. We shall work on $\vec{x_u}$, $0 \leq u \leq k$, simultaneously in a pipeline. The random variable tree for P_u (except that for P_0) is not constructed before the derandomization process begins, rather it is constructed from a forest as the derandomization process proceeds. A forest containing 2^u random variable trees corresponds to each variable x_{i_u} in P_u because there are 2^u bit patterns for x_{i_j}, $0 \le j < u$. We use F_u to denote the random variable forest for P_u. We are to fix the random bits on the l-th level of F_v (for $\vec{x_v}$) under the condition that random bits from level 0 to level $l+c-1$, $0 \le c \le v$, in F_{v-c} have already been fixed. We are to perform this fixing in constant time. The 2^u random variable trees corresponding to each random variable x_{i_u} are built bottom up as the derandomization process proceeds. Immediately before the step we are to fix the random bits on the l-th level of F_u, the 2^u random variable trees corresponding to x_{i_u} are constructed up to the l-th level. The details of the algorithm for constructing the random variable trees will be given later in this section.

Consider a GPC function $f_{i,j}(x_i, x_j)$ under the condition stated in the last paragraph. When we start working on $\vec{x_v}$ we should have the BPC functions $f_{i_v,j_v}(x_{i_v}, x_{j_v})$ evaluated and stored in a table. However, because $\vec{x_u}$, $0 \le u < v$, have not been fixed yet, we have to try out all possible situations. There are a total of 4^v patterns for bits x_{i_u}, x_{j_u}, $0 \le u < v$, we use 4^v BPC functions for each pair (i,j). By $f_{i_v,j_v}(x_{i_v}, x_{j_v})(y_{v-1}y_{v-2}\cdots y_0, z_{v-1}z_{v-2}\cdots z_0)$ we denote the function $f_{i_v,j_v}(x_{i_v}, x_{j_v})$ obtained under the condition that $(x_{i_{v-1}}x_{i_{v-2}}\cdots x_{i_0}, x_{j_{v-1}}x_{j_{v-2}}\cdots x_{j_0})$ is set to $(y_{v-1}y_{v-2}\cdots y_0, z_{v-1}z_{v-2}\cdots z_0)$.

For each pair $(w, w\#0)$ at each level l (this is the level in the random variable forest), $0 \le l \le \log n$, a *bit pipeline tree* is built (Fig. 3) which is a complete binary tree of height $2k$. Nodes at even depth from the root in a bit pipeline tree are selectors, nodes at odd depth are fanout gates. A signal *true* is initially input into the root of the tree and propagates downward toward the leaves. The selectors at depth $2d$ select the output by the decision of the random bits which are the parents of random variables $x_{w_d}, x_{w\#0_d}$ in F_d. There is one random variable corresponds to each selector. Let random variable r corresponds to the selector s. If r is set to 0 then s selects the left child and propagates the true signal to its left child while no signal is sent to its right child. If r is set to 1 then the true signal will be sent to the right child and no signal will be sent to the left child. If s does not receive any signal from its parent then no signal will be propagated to s's children no matter how r is set. The gates at odd depth in the bit pipeline tree are fanout gates and pointers from them to their children are labeled with bits which are conditionally set. Refer to Fig. 3 which shows a bit pipeline tree of height 4. If the selector at the root (node 0) selects 0 (which means that the random variable which is the parent of x_{w_0} and $x_{w\#0_0}$ in the random variable forest is set to 0), then $x_{w_0} = x_{w\#0_0}$, therefore the two random variables can only assume the patterns 00 or 11 which are labeled on the pointers from node 1. If, on the other hand, node 0 selects 1 then $x_{w_0} = 1 - x_{w\#0_0}$, the two random variables can only assume the patterns 01 or 10 which are labeled on the pointers of node 2. Let us take node 4 as another example. If node 4 selects 0 then $x_{w_1} = x_{w\#0_1}$, thus the pointers of node 9 are labeled with $\begin{matrix}1 & 1\\ 0 & 0\end{matrix}$ and $\begin{matrix}1 & 1\\ 1 & 1\end{matrix}$. This indicates that the bits for $(w_1w_0, w\#0_1w\#0_0)$ can have two patterns, $(01, 01)$ or $(11, 11)$.

The bit pipeline tree built for level $\log n$ has height k. No fanout gates will be used. This is a special and simpler case compared to the bit pipeline trees for other levels. In the following discussion we only consider bit pipeline tree for levels other than $\log n$.

Lemma 5: In a bit pipeline tree there are exactly 2^d nodes at depth $2d$ which will receive the true signal from the root.

Proof: Each selector selects only one path. Each fanout gate sends the true signal to both children. Therefore exactly 2^d nodes at depth $2d$ will receive the true signal from the root. □

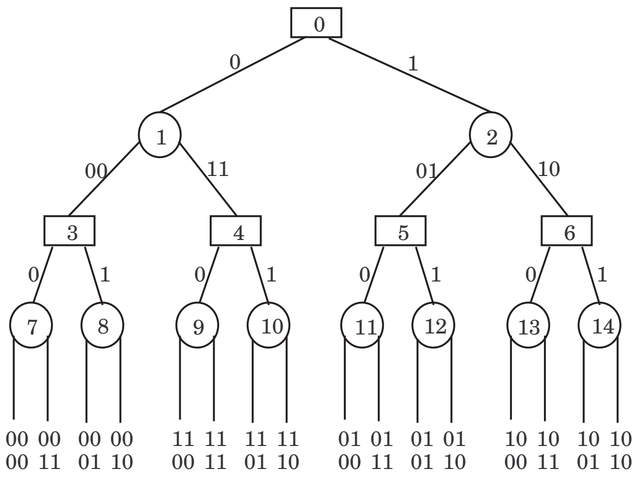

Fig. 3. A bit pipeline tree of height 4

For each node i at even depth we shall also say that it has the *conditional bit pattern* (or *conditional bits*, *bit pattern*) which is the pattern labeled on the pointer from $p(i)$. The root of the bit pipeline tree has empty string as its bit pattern.

Define step 0 as the step when the true signal is input to node 0. The function of a bit pipeline tree can be described as follows.

Step t: Selectors at depth $2t$ which have received true signal selects 0 or 1 for $(w_t, w\#0_t)$. Pass the true signal and the bit setting information to nodes at depth $2t+2$.

Now consider the selectors at depth $2d$. By Lemma 5 a set of 2^d selectors at depth $2d$ receive the true signal. We call this set the *surviving set* $S^l_{w,d}$. We also denote by $S^l_{w,d}$ the set of bit patterns the 2^d surviving selectors have, where w in the subscript is for $(w, w\#0)$ and l is the level for which the pipeline tree is built. Let selector $s \in S^l_{w,d}$ have bit pattern $(y_{d-1}y_{d-2}\cdots y_0, z_{d-1}z_{d-2}\cdots z_0)$. s compares

$$f^{(l)}_{w_d,w\#0_d}(0,0)(y_{d-1}y_{d-2}\cdots y_0, z_{d-1}z_{d-2}\cdots z_0)+$$
$$f^{(l)}_{w_d,w\#0_d}(1,1)(y_{d-1}y_{d-2}\cdots y_0, z_{d-1}z_{d-2}\cdots z_0)+$$
$$f^{(l)}_{w\#0_d,w_d}(0,0)(z_{d-1}z_{d-2}\cdots z_0, y_{d-1}y_{d-2}\cdots y_0)+$$
$$f^{(l)}_{w\#0_d,w_d}(1,1)(z_{d-1}z_{d-2}\cdots z_0, y_{d-1}y_{d-2}\cdots y_0)$$

with

$$f^{(l)}_{w_d,w\#0_d}(0,1)(y_{d-1}y_{d-2}\cdots y_0, z_{d-1}z_{d-2}\cdots z_0)+$$
$$f^{(l)}_{w_d,w\#0_d}(1,0)(y_{d-1}y_{d-2}\cdots y_0, z_{d-1}z_{d-2}\cdots z_0)+$$
$$f^{(l)}_{w\#0_d,w_d}(0,1)(z_{d-1}z_{d-2}\cdots z_0, y_{d-1}y_{d-2}\cdots y_0)+$$
$$f^{(l)}_{w\#0_d,w_d}(1,0)(z_{d-1}z_{d-2}\cdots z_0, y_{d-1}y_{d-2}\cdots y_0)$$

and selects 0 if former is no less than the latter and selects 1 otherwise. Note that the selectors which do not receive the true signal (there are $4^d - 2^d$ of them) have bit patterns which are eliminated.

Let $LS^l_{w,d} = \{\alpha | (\alpha,\beta) \in S^l_{w,d}\}$ and $RS^l_{w,d} = \{\beta | (\alpha,\beta) \in S^l_{w,d}\}$.

Lemma 6: $LS^l_{w,d} = RS^l_{w,d} = \{0,1\}^d$.

Proof: By induction. Assuming that it is true for bit pipeline trees of height $2d-2$. A bit pipeline tree of height $2d$ can be constructed by using a new selector as the root, two new fanout gates at depth 1, and four copies of the bit pipeline tree of height $2d-2$ at depth 2. If the root selects 0 then patterns 00 and 11 are concatenated with patterns in $S^l_{w,d-1}$, therefore both $LS^l_{w,d-1}$ and $RS^l_{w,d-1}$ are concatenated with $\{0,1\}$. The situation when the root selects 1 is similar. $\Box$

Now let us consider how functions $f^{(l)}_{i_d,j_d}(x_{i_d}, x_{j_d})(\alpha,\beta)$ are combined. Take the difficult case where both i and j are odd. By Lemma 6 there is only one pattern $p_1 = (\alpha',\alpha) \in S^l_{i\#0,d}$ and there is only one pattern $p_2 = (\beta',\beta) \in S^l_{j\#0,d}$. If the selector having bit pattern p_1 selects 0 then $x_{i_d} = x_{i\#0_d}$ else $x_{i_d} = 1 - x_{i\#0_d}$. If the selector having bit pattern p_2 selects 0 then $x_{j_d} = x_{j\#0_d}$ else $x_{j_d} = 1 - x_{j\#0_d}$. In any case the conditional bit pattern is changed to (α',β'), *i.e.*, $f^{(l)}_{i_d,j_d}(x_{i_d}, x_{j_d})(\alpha,\beta)$ will be combined into $f^{(l+1)}_{\lfloor i/2\rfloor_d,\lfloor j/2\rfloor_d}(x_{\lfloor i/2\rfloor_d}, x_{\lfloor j/2\rfloor_d})(\alpha',\beta')$. Note that $x_{\lfloor i/2\rfloor_d}$ and $x_{\lfloor j/2\rfloor_d}$ are new random variables and here we are not using supscript to denote this fact. The following lemma ensures that at most four functions will be combined into $f^{(l+1)}_{\lfloor i/2\rfloor_d,\lfloor j/2\rfloor_d}(x_{\lfloor i/2\rfloor_d}, x_{\lfloor j/2\rfloor_d})(\alpha',\beta')$.

Let $S = \{(\alpha',\beta') | (\alpha',\alpha) \in S^l_{i,d}, (\beta',\beta) \in S^l_{j,d}, \alpha,\beta \in \{0,1\}^d\}$.

Lemma 7: $|S| = 4^d$.

Proof: The definition of S can be viewed as a linear transformation. Represent $x \in \{0,1\}^d$ by a vector of 2^d bits with x-th bit set to 1 and the rest of the bits set to 0. The transformation $\alpha \mapsto \alpha'$ can be represented by a permutation matrix of order 2^d. The transformation $(\alpha,\beta) \mapsto (\alpha',\beta')$ can be represented by a permutation matrix of order 2^{d+1}. $\Box$

Lemma 7 tells us that the functions to be combined are permuted, therefore no more than four functions will be combined under any conditional bit pattern.

We call this scheme of combining as *combining functions with respect to the surviving set.*

We have completed a preliminary description of our derandomization scheme for the GPC problem. The algorithm for processors working on $\vec{x_d}$, $0 \le d < k$, can be summarized as follows.

Step t ($0 \le t < d$): Wait for the pipeline to be filled.

Step $d+t$ ($0 \le t < \log n$): Fix random variables at level t for all conditional bit patterns in the surviving set. (* There are 2^d such patterns in the surviving set.*) Combine functions with respect to the surviving set. (* At the same time the bit setting information is transmitted to the nodes at depth $2d+2$ on the bit pipeline tree. *)

Step $d+\log n$: Fix the only remaining random variable at level $\log n$ for the only bit pattern in the surviving set. Output the good point for $\vec{x_d}$. (* At the same time the bit setting information is transmitted to the node at depth $d+1$ on the bit pipeline tree.*)

Theorem 8: The GPC problem can be solved on the CREW PRAM in time $O((q/k+1)(\log n + \tau))$ with $O(4^k m)$ processors, where τ is the time for computing the BPC functions $f_{i_d,j_d}(x_{i_d}, x_{j_d})(\alpha,\beta)$.

Proof: The correctness of the scheme comes from the fact that as random bits are fixed a smaller space with higher expectation is obtained, and thus when all random bits are fixed a good point is found. Since k $\vec{x_u}$'s are fixed in one pass which takes $O(\log n + \tau)$ time, the time complexity for solving the GPC problem is $O((q/k + 1)(\log n + \tau))$. The processor complexity is obvious from the description of the scheme. $\Box$

We have not yet discussed explicitly the way the random variable trees are constructed. The construction is implied in the surviving set we computed. We now give the algorithm for constructing the random variable trees. This algorithm will help understand better the whole scheme.

The i-th node under conditional bit pattern j at the l-th level of the random variable trees for P_u is stored in $T_u^{(l)}[i][j]$. The leaves are stored in $T_u^{(-1)}$. Initially bit pipeline trees for level -1 are built such that $T_u^{(-1)}[i][j]$ has two children $T_{u+1}^{(-1)}[i][j0]$, $T_{u+1}^{(-1)}[i][j1]$, where $j0$ and $j1$ are the concatenations of j with 0 and 1 respectively. Note that the bit pipeline tree constructed here is different from the one we built before, but in principle they are the same tree and perform the same function in our scheme. The algorithm for constructing the random variable trees for P_u is below.

```
Procedure **RV-Tree**

**begin**

Step $t$ $(0 \le t < u)$: Wait for the pipeline to be filled.

Step $u + t$ $(0 \le t < \log n)$:

(* In this step we are to build $T_u^{(t)}[i][j]$, $0 \le i < n/2^{t+1}$, $0 \le j < 2^u$. At the beginning of this step $T_{u-1}^{(t)}[i][j]$ has already been constructed. Let $T_{u-1}^{(t-1)}[i0][j]$ and $T_{u-1}^{(t-1)}[i1][j']$ be the two children of $T_{u-1}^{(t)}[i][j]$ in the random variable tree. $T_u^{(t-1)}[i0][j0]$ and $T_u^{(t-1)}[i0][j1]$ are the children of $T_{u-1}^{(t-1)}[i0][j]$, $T_u^{(t-1)}[i1][j'0]$ and $T_u^{(t-1)}[i1][j'1]$ are the children of $T_{u-1}^{(t-1)}[i1][j']$, in the bit pipeline tree for level $t-1$. The setting of the random variable $r$ for the pair $(i0, i1)$ at level $t$ for $P_{u-1}$, *i.e.* the random variable in $T_{u-1}^{(t)}[i][j]$, is known. *)

make $T_u^{(t-1)}[i0][j0]$ and
$T_u^{(t-1)}[i1][j'r]$ as the children of
$T_u^{(t)}[i][j0]$ in the random variable forest for $P_u$;
(* $jr$ is the concatenation of $j$ and $r$. *)

make $T_u^{(t-1)}[i0][j1]$ and
$T_u^{(t-1)}[i1][j'\overline{r}]$ as the children of
$T_u^{(t)}[i][j1]$ in the random variable forest for $P_u$;
(* $\overline{r}$ is the complement of $r$. *)

make $T_u^{(t)}[i][j0]$ and
$T_u^{(t)}[i][j1]$ as the children of
$T_{u-1}^{(t)}[i][j]$ in bit pipeline tree for level $t$;
```

```
Fix the random variables in $T_u^{(t)}[i][j0]$ and $T_u^{(t)}[i][j1]$;

Step $u + \log n$:

(* At the beginning of this step the random variable trees have been built for $T_i$, $0 \le i < u$. Let $T_{u-1}^{(\log n)}[0][j]$
be the root of $T_{u-1}$. The random variable $r$ in $T_{u-1}^{(\log n)}[0][j]$ has been fixed. In this step we are to choose one
of the two children of $T_{u-1}^{(\log n)}[0][j]$ in the bit pipeline tree for level $\log n$ as the root of $T_u$. *)

make $T_u^{(\log n-1)}[0][jr]$ as the child of
$T_u^{(\log n)}[0][jr]$ in the random variable tree;

make $T_u^{(\log n)}[0][jr]$ as the child of
$T_{u-1}^{(\log n)}[0][j]$ in the bit pipeline tree for level $\log n$;

fix the random variable in $T_u^{(\log n)}[0][jr]$;

output $T_u^{(\log n)}[0][jr]$ as the root of $T_u$;

end
```

Procedure RV-Tree uses the pipelining technique as well as the dynamic programming technique. These are some of the essential elements of our scheme.

We now show how to remove the concurrent read feature from the scheme. The difficulty here is in the step of combining functions with respect to the surviving set. The size of the surviving set $S_{u,k}^l$ is 2^k while there are 4^k conditional bit patterns. There are 4^k functions $f_{u_k,v_k}^{(l)}(x_{u_k},\ x_{v_k})$, one for each bit pattern $(\alpha,\ \beta)$. All 4^k functions will consult the surviving set in order for them to be combined into new functions. The problem is how to do it in constant time without resorting to concurrent read.

We show how to let $f_{u_k,u\#0_k}^{(l)}(x_{u_k}, x_{u\#0_k})(\alpha,\beta)$ to acquire the bit pattern α' which satisfies $(\alpha',\beta) \in S_{u,k}^l$. Function $f_{u_k,v_k}^{(l)}(x_{u_k}, x_{v_k})(\alpha,\beta)$ can then obtain the bit pattern α' from $f_{u_k,u\#0_k}^{(l)}(x_{u_k}, x_{u\#0_k})(\alpha,\beta)$ by the pipeline scheme described in [H3].

Suppose we are to solve P_u, $0 \le u \le k$, in one pass. We solve 4^k copies of P_{k-1}, one copy corresponds to one conditional bit pattern in P_k. $f_{u_k,v_k}^{(l)}(x_{u_k}, x_{v_k})(\alpha,\beta)$ in P_k can obtain α' by following the computation in the copy of P_{k-1} which corresponds to (α,β). This can be done without concurrent read. Now for each of the 4^k copies of P_{k-1} we solved 4^{k-1} copies of P_{k-2}, one copy corresponds to one conditional bit pattern in P_{k-1}. And so on. Thus to remove concurrent read we need $c^{k^2}(m+n)$ processors for solving P_u, $0 \le u \le k$, in one pass, where c is a suitable constant. Note also that it takes $O(k^2)$ time to make needed copies.

Theorem 9: The general pairs PROFIT/COST problem can be solved on the EREW PRAM in time $O((q/\sqrt{k}+1)(\log n + k + \tau))$ with $O(c^k m)$ processors, where c is a suitable constant and τ is the time for computing the bit pairs PROFIT/COST functions $f_{u_d,v_d}(x_{u_d}, x_{v_d})(\alpha,\beta)$. □

3.2 $O(\log n)$-wise Independent Random Variables

We consider the derandomization of functions of the form $B(\vec{x}) = \sum_i^{n^a} f_i(x_{i,1}, x_{i,2}, ..., x_{i,b\log n})$, where a, b are constants and $x_{i,j}$'s are 0/1-valued uniformly distributed mutually independent random variables. The problem is to find a good point $\vec{y}$ such that $B(\vec{y}) \geq E[B(\vec{x})]$. The derandomization algorithm for this problem is given by Berger and Rompel[BR] which is a generalization of Luby's algorithm[L2].

Since each function $f_i(x_{i,1}, x_{i,2}, ..., x_{i,b\log n})$ contains at most $b\log n$ random variables, if every $b\log n$ random variables among the $x_{i,j}$'s are mutually independent then the value of $E[B(\vec{x})]$ would be the same as that when all $x_{i,j}$'s are mutually independent. A sample space containing n mutually independent random variables has $\Omega((n/k)^{\lceil k/2 \rceil})$ sample points[ABI]. A sample space containing n 0/1-valued uniformly distributed $(b\log n)$-wise independent random variables can be constructed with $O(n^{\log n})$ sample points.

The sample space is thus constructed by taking n $(b\log n)$-wise linearly independent binary vectors $a_1, a_2, ..., a_n$, each of length $l = O(\log^2 n)$. The value of $a_i =< a_{i,1}, a_{i,2}, ..., a_{i,l} >$ at sample point $r =< r_1, r_2, ..., r_l >$ is $(\sum_{k=1}^{l} a_{i,k} r_k)$ mod 2. Using elementary linear algebra it is easy to show that a_i's are $(b\log n)$-wise independent random variables.

The problem now is to find a good point r' such that $B(\vec{x}(r')) \geq E[B(\vec{x}(r))]$.

To determinize r, one bit of r will be fixed at a time. Fixing one bit r_i of r is to partition the sample space to two subspaces, one is the subspace in which $r_i = 0$ and the other is the subspace in which $r_i = 1$, and to determine which subspace will be discarded. In the remaining subspace the value of r_i is fixed. Thus the fixing of binary bits can be viewed as a binary search into the sample space.

Assuming that $r_1 = s_1, ..., r_{t-1} = s_{t-1}$ have already been set, we compute $E[B(\vec{x})|r_1 = s_1, r_2 = s_2, ..., r_{t-1} = s_{t-1}, r_t = 0]$ and $E[B(\vec{x})|r_1 = s_1, r_2 = s_2, ..., r_{t-1} = s_{t-1}, r_t = 1]$. r_t is set to $s_t \in \{0, 1\}$ which maximizes $E[B(\vec{x})|r_1 = s_1, r_2 = s_2, ..., r_{t-1} = s_{t-1}, r_t = s_t]$. Assume that $E[B(\vec{x})|r_1 = s_1, r_2 = s_2, ..., r_{t-1} = s_{t-1}] \geq E[B(\vec{x})]$. We have $E[B(\vec{x})|r_1 = s_1, r_2 = s_2, ..., r_{t-1} = s_{t-1}, r_t = s_t] = \max\{E[B(\vec{x})|r_1 = s_1, r_2 = s_2, ..., r_{t-1} = s_{t-1}, r_t = 0], E[B(\vec{x})|r_1 = s_1, r_2 = s_2, ..., r_{t-1} = s_{t-1}, r_t = 1]\}$
$\geq (E[B(\vec{x})|r_1 = s_1, r_2 = s_2, ..., r_{t-1} = s_{t-1}, r_t = 0] + E[B(\vec{x})|r_1 = s_1, r_2 = s_2, ..., r_{t-1} = s_{t-1}, r_t = 1])/2$
$= E[B(\vec{x})|r_1 = s_1, r_2 = s_2, ..., r_{t-1} = s_{t-1}] \geq E[B(\vec{x})]$.

Berger and Rompel gave the following method for evaluating conditional expectations. Since function B is the sum of function f_i's, the conditional expectation can be evaluated on each f_i in parallel and then the conditional expectation of B can be obtained from the sum of the conditional expectations of f_i's. Consider the problem of evaluating $E[f_i(x_{i,1}, ..., x_{i,b\log n})|r_1 = s_1, ..., r_t = s_t]$. Let x be the vector $< x_{i,1}, ..., x_{i,b\log n} >$ and A be the matrix whose j-th row is $a_{i,j}$, the binary vector of length l corresponding to $x_{i,j}$. Then $x = Ar$, and

$$E[f_i(x_{i,1}, ..., x_{i,b\log n})|r_1 = s_1, ..., r_t = s_t] = \sum_x f_i(x) Pr[Ar = x|r_1 = s_1, ..., r_t = s_t].$$

Let $r' =< r_1, r_2, ..., r_t >$, $r'' =< r_{t+1}, ..., r_l >$, $s =< s_1, ..., s_t >$, A' and A'' be the first t and the last $l - t$ columns of A respectively, then

$$\sum_x f_i(x)Pr[Ar = x|r_1 = s_1, ..., r_t = s_t] = \sum_x f_i(x)Pr[A'r' + A''r'' = x|r' = s] = \sum_x f_i(x)Pr[A''r'' = x - A's].$$

If $A''r'' = x - A's$ is solvable, then $Pr[A''r'' = x - A's] = 2^{-\text{rank}(A'')}$, otherwise $Pr[A''r'' = x - A's] = 0$.

The above scheme allow us to compute the conditional expectations in polylogarithmic time using a polynomial number of processors.

4 Applications

The idea of derandomization has been successfully applied to the design of several efficient sequential and parallel algorithms [ABI] [BR] [BRS] [H3] [H4] [HI] [KW] [L1] [L2] [L3] [MNN] [PSZ] [Rag] [Sp]. Spencer[Sp] and Raghavan[Rag] first used a binary search technique to locate a good sample point. Their binary search technique enables an algorithm to search an exponential sized sample space in polynomial time. Efficient sequential algorithms have been obtain by using this technique[Rag] [Sp].

In the design of efficient parallel algorithms, Karp and Wigderson[KW], Luby[L1] and Alon *et al.*[ABI] designed small sample space by using limited independence. Since the sample space they designed contains a polynomial number of points, exhaustive search were used to locate a good point to obtain a DNC algorithm. This technique has been used successfully by Karp and Wigderson[KW] and Luby[L1] to obtain DNC algorithms for the maximal independent set problem. By applying this technique Alon *et al.*[ABI] obtained DNC algorithms for several problems, among them finding an independent set of size k for a hypergraph, finding a large d-partite subhypergraph, Siden-subsets of graphs and Ramsey-type problems.

In order to obtain processor efficient parallel algorithms through derandomization, Luby[L2] [L3] used the idea of binary search on a small sample space. His idea yields efficient parallel algorithms for the $\Delta + 1$ vertex coloring problem, the maximal independent set problem and the maximal matching problem. His DNC algorithms for the maximal independent set problem and the maximal matching problem have time complexity which are fairly close to the time complexity of the algorithms for the two problems obtained through ad hoc designs[GS1] [GS2] [IS].

Luby's technique was carried further by Berger and Rompel[BR] and Motwani *et al.* [MNN] where $(\log^c n)$-wise independence among random variables was used. Because the sample space contains $n^{\log^{O(1)} n}$ sample points in the design of [BR] [MNN], they used a thoughtfully designed binary search technique to obtain DNC algorithms. They presented DNC algorithms for the set discrepancy problem and the hypergraph coloring problem.

Recently Han and Igarashi[H3] [HI] showed how to obtain an efficient parallel algorithm for the bit pairs PROFIT/COST problem through derandomization using an exponential sized sample space. Their derandomization scheme gives a case in which the sample space can be reduced at the rate of $\sqrt{n}$. This technique can be developed further to obtain fast and efficient parallel algorithms for the $\Delta + 1$ vertex coloring problem, the maximal independent set problem and the maximal matching problem[H4].

In the rest of this section we will show several applications of derandomization schemes.

4.1 Large d-Partite Subhypergraph

A hypergraph $H = (V, E)$ contains a set of vertices V and a set of edges E. Each edge is a subset of V. H is d-uniform if every edge has d elements. Given a d-uniform hypergraph H with $d \geq 2$, we intend to find a partition $(V_1, V_2, ..., V_d)$ of V such that the number of edges of H having precisely one vertex in each class V_i is at lease $\lfloor |E|d!/d^d \rfloor$. We show the design of a deterministic parallel algorithm for this problem through derandomization. This design is from [ABI].

Let c_i, $1 \leq i \leq |V|$, be d-wise independent random variables uniformly distributed on $\{1, 2, ..., d\}$. Color vertex i with color c_i. A given edge is good if it has all the colors. The probability that a given edge is good is $d!/d^d$. Thus the expected number of good edges is $|E|d!/d^d$.

To obtain $|V|$ random variables which are d-wise independent and distributed almost equally on $\{1, 2, ..., d\}$, we pick a prime $q > 2d^2|E|$ and construct a sample space as given in section 2.1. The probability that a given edge is good is greater than

$$d!(\frac{1}{d} - \frac{1}{2d^2|E|})^d > \frac{d!}{d^d}(1 - \frac{1}{2|E|}).$$

The expected number of good edges is greater than $|E|d!/d^d - 1/2$. That is, the expected number of good edges is at least $\lfloor |E|d!/d^d \rfloor$.

The sample space contains q^d sample points. Thus when d is a constant a polynomial number of processors is sufficient to perform an exhaustive search on all sample points and a NC algorithm can be derived.

4.2 The Vertex Partitioning Problem

The vertex partitioning problem[L2] is to label vertices of a graph $G = (V, E)$ with 0's and 1's such that the number of edges incident with both 0 and 1 labeled vertices is at least half of the total number of edges. This is a special case of the d-partite subhypergraph problem we discussed in the last subsection. Here we may use n 0/1-valued uniformly distributed mutually independent random variables, one for each vertex of the graph. Let $f(x_i, x_j) = x_i \oplus x_j$ and $F = \sum_{(i,j)\in E} f(x_i, x_j)$. The value of F is the number of edges incident with both 0 and 1 labeled vertices. Moreover, $E[F] = \sum_{(i,j)\in E}(f(0,0)+f(0,1)+f(1,0)+f(1,1))/4 = |E|/2$. Therefore there exists a sample point p such that $F(p) \geq |E|/2$. Function F is the BENEFIT function of the bit pairs PROFIT/COST problem. Thus for the case $d = 2$ the d-partite hypersubgraph problem can be solved by our fast derandomization algorithm for the bit pairs PROFIT/COST problem.

4.3 Independent Set on Graphs

An independent set of a graph is a set I of vertices such that no two vertices in I are adjacent. I is maximal if $I \subseteq J$ and J is an independent set implies that $J = I$. For certain restricted graphs a maximal independent set can be found by a parallel algorithm runs in less than logarithmic time[H1] [H2]. We shall give a deterministic algorithm[H4] for computing a maximal independent set for general graphs through the derandomization of a randomized algorithm due to Luby[L3].

Currently the fastest randomized algorithm for computing a maximal independent set is due to Luby[L1] which runs in expected $O(\log n)$ time using a linear number of processors. This algorithm uses a large sample space and no efficient derandomization scheme is known to derandomize it. We will present another randomized algorithm, also due to Luby[L1], which runs in $O(\log^2 n)$ time.

Let $G = (V, E)$ be a graph. Let $d(i)$ be the degree of vertex i in G. Let k_i be such that $2^{k_i-1} < 2d(i) \leq 2^{k_i}$. Let $q = \max\{k_i | i \in V\}$. Let $\vec{x} = < x_i \in \{0,1\}^q, i \in V\}$. Each x_i is a random variable uniformly distributed on $\{0, 1, ..., 2^q - 1\}$. x_i is associated with vertex i. Define

$$Y_i(x_i) \quad = \quad \begin{cases} 1 & \text{if } x_i(k_i - 1) \cdots x_i(0) = 1^{k_i} \\ 0 & \text{otherwise} \end{cases}$$

where $x_i(p)$ is the p-th bit of x_i.

Luby used the following randomized algorithm to find an independent set I such that an expected constant fraction of the edges will be incident with vertices in $I \cup N(I)$, where $N(I) = \{j \in V | \exists i \in I, (i, j) \in E\}$.

```
Procedure **Independent**
begin
    $I := \phi$;
    In parallel for all $i$ such that $d(i) = 0$ do
        $I := I \cup \{i\}$;

    In parallel for all $i$ such that $d(i) \neq 0$ do
        generate random number $x_i$;
        if $Y_i(x_i) = 1$ then $I := I \cup \{i\}$;

    In parallel for all $(i, j) \in E$ do
        if $i \in I$ and $j \in I$ then
        if $d(i) \leq d(j)$ then $I := I - \{i\}$;
        else $I := I - \{j\}$;
end
```

Through rather complicated probabilistic analysis[L1] Luby showed that after an execution of the above algorithm the expected number of edges deleted is at least a constant fraction of the number of edges in the input graph. The probabilistic analysis uses only the pairwise independence of the random variables. Procedure Independent can be executed in $O(\log n)$ time on an EREW PRAM using a linear number of processors[L1]. A maximal independent set can be computed by the following procedure.

```
Procedure **Max-Independent**
begin
    $I := \phi$;
    $V' := V$;
    while $V' \neq \phi$ do
        begin
```

```
            Find an independent set $I' \subseteq V'$ using procedure Independent;
            $I := I \cup I'$;
            $V' := V' - (I' \cup N(I'))$;
        **end**
**end**
```

We will use a fast derandomization scheme to derandomize procedure Independent to obtain a deterministic parallel algorithm.

A vertex j will be in I after an execution of Independent if $Y_j(x_j) = 1$ and $Y_k(x_k) = 0$ for any neighbor k of j such that $d(k) \geq d(j)$. Define $Y_{j,k}(x_j, x_k) = -Y_j(x_j)Y_k(x_k)$. If $Y_j(x_j) + \sum_{k \in adj(j), d(k) \geq d(j)} Y_{j,k}(x_j, x_k) = 1$ then $j \in I$ and if it is ≤ 0 then $j \notin I$. A vertex i will be in $N(I)$ after an execution of Independent if any one of the neighbors of i is in I. Therefore if

$$\sum_{j \in adj(i)} \left(Y_j(x_j) + \sum_{k \in adj(j), d(k) \geq d(j)} Y_{j,k}(x_j, x_k) + \sum_{k \in adj(i) - \{j\}} Y_{j,k}(x_j, x_k) \right)$$

is 1 then $i \in N(I)$, if it is ≤ 0 then i may or may not be in $N(I)$. Now the following BENEFIT function gives a lower bound on the number of edges deleted after an execution of Independent[L1].

$$B(\vec{x}) = \sum_{i \in V} \frac{d(i)}{2} \sum_{j \in adj(i)} \left(Y_j(x_j) + \sum_{k \in adj(j), d(k) \geq d(j)} Y_{j,k}(x_j, x_k) + \sum_{k \in adj(i) - \{j\}} Y_{j,k}(x_j, x_k) \right)$$

$\frac{d(i)}{2}$ is used because each edge has been counted twice at the two vertices it is incident with. The above formulation is due to Luby[L1] [L3] and he also showed that the expected value of B is $\geq |E|/c$ for a constant $c > 1$.

Function B can be written as,

$$\begin{aligned} B(\vec{x}) = & \sum_{j \in V} (\sum_{i \in adj(j)} \frac{d(i)}{2}) Y(x_j) + \sum_{(j,k) \in E, d(k) \geq d(j)} (\sum_{i \in adj(j)} \frac{d(i)}{2}) Y_{j,k}(x_j, x_k) \\ & + \sum_{i \in V} \frac{d(i)}{2} \sum_{j,k \in adj(i), j \neq k} Y_{j,k}(x_j, x_k) \end{aligned}$$

It is now not difficult to see that we can obtain the following standard form of the BENEFIT function B by applying a matrix multiplication and then combining functions of the same subscript,

$$B(\vec{x}) = \sum_i f_i(x_i) + \sum_{(i,j)} f_{i,j}(x_i, x_j)$$

.

Such a matrix multiplication can be done in $O(\log n)$ time using $n^{2.376}$ processors[CW]. By Theorem 8 the GPC problem can be solved in $O(\log n)$ time if we have $n^{2.376}$ processors. This eliminates a constant fraction of the edges from the graph. Therefore a maximal independent set can be found with $O(\log n)$ iterations of such processing.

Theorem 10: A maximal independent set of a graph can be computed in $O(\log^2 n)$ time with $n^{2.376}$ processors on the CREW PRAM. □

More sophisticated application of the derandomization scheme yields fast and efficient parallel algorithms for the maximal independent set problem, the maximal matching problem and the $\Delta + 1$ vertex coloring problem. These results are given in [H4].

Applications of derandomization techniques to the design of several important parallel algorithms can be found in [ABI] [BR] [BRS] [H3] [H4] [HI] [KW] [L1] [L2] [L3] [MNN] [PSZ].

References

[AKS] M. Ajtai, J. Komlós and E. Szemerédi. An $O(N \log N)$ sorting network. Proc. 15th ACM Symp. on Theory of Computing, 1-9(1983).

[ABI] N. Alon, L. Babai, A. Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. J. of Algorithms 7, 567-583(1986).

[BR] B. Berger, J. Rompel. Simulating $(\log^c n)$-wise independence in NC. Proc. 30th Symp. on Foundations of Computer Science, IEEE, 2-7(1989).

[BRS] B. Berger, J. Rompel, P. Shor. Efficient NC algorithms for set cover with applications to learning and geometry. Proc. 30th Symp. on Foundations of Computer Science, IEEE, 54-59(1989).

[C] R. Cole. Parallel merge sort. Proc. 27th Symp. on Foundations of Computer Science, IEEE, 511-516(1986).

[Co1] S. A. Cook. A taxonomy of problems with fast parallel algorithms. Information and Control, Vol. 64, Nos. 1-3, 1985.

[Co2] S. A. Cook. Deterministic CFL's are accepted simultaneously in polynomial time and log square space. 1979 Proceedings of ACM STOC, pp. 338-345.

[CW] D. Coppersmith, S. Winograd. Matrix multiplication via arithmetic progressions. Proc. 19th Ann. ACM Symp. on Theory of Computing, 1-6(1987).

[FW] S. Fortune and J. Wyllie. Parallelism in random access machines. Proc. 10th Annual ACM Symp. on Theory of Computing, San Diego, California, 1978, 114-118.

[GS1] M. Goldberg, T. Spencer. A new parallel algorithm for the maximal independent set problem. SIAM J. Comput., Vol. 18, No. 2, pp. 419-427(April 1989).

[GS2] M. Goldberg, T. Spencer. Constructing a maximal independent set in parallel. SIAM J. Dis. Math., Vol 2, No. 3, 322-328(Aug. 1989).

[H1] Y. Han. Matching partition a linked list and its optimization. Proc. 1989 ACM Symposium on Parallel Algorithms and Architectures, Santa Fe, New Mexico, 246-253(June, 1989).

[H2] Y. Han. Parallel algorithms for computing linked list prefix. J. of Parallel and Distributed Computing 6, 537-557(1989).

[H3] Y. Han. A parallel algorithm for the PROFIT/COST problem. Proc. of 1991 Int. Conf. on Parallel Processing, Vol. III, 107-114, St. Charles, Illinois.

[H4] Y. Han: A fast derandomization scheme and its applications. Proc. 1991 Workshop on Algorithms and Data Structures, Ottawa, Canada, Lecture Notes in Computer Science 519, 177-188(August 1991). Full version in TR No. 180-90, Dept. Computer Sci., Univ. of Kentucky.

[HI] Y. Han and Y. Igarashi. Derandomization by exploiting redundancy and mutual independence. Proc. Int. Symp. SIGAL'90, Tokyo, Japan, LNCS 450, 328-337(1990).

[IS] A. Israeli, Y. Shiloach. An improved parallel algorithm for maximal matching. Information Processing Letters 22(1986), 57-60.

[Jo] A. Joffe. On a set of almost deterministic k-independent random variables. Ann. Probability 2 (1974), 161-162.

[KW] R. Karp, A. Wigderson. A fast parallel algorithm for the maximal independent set problem. JACM 32:4, Oct. 1985, 762-773.

[L1] M. Luby. A simple parallel algorithm for the maximal independent set problem. SIAM J. Comput. 15:4, Nov. 1986, 1036-1053.

[L2] M. Luby. Removing randomness in parallel computation without a processor penalty. Proc. 29th Symp. on Foundations of Computer Science, IEEE, 162-173(1988).

[L3] M. Luby. Removing randomness in parallel computation without a processor penalty. TR-89-044, Int. Comp. Sci. Institute, Berkeley, California.

[MNN] R. Motwani, J. Naor, M. Naor. The probabilistic method yields deterministic parallel algorithms. Proc. 30th Symp. on Foundations of Computer Science, IEEE, 8-13(1989).

[MR] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. Proc. 26th Symp. on Foundations of Computer Science, IEEE, 478-489(1985).

[PSZ] G. Pantziou, P. Spirakis, C. Zaroliagis. Fast parallel approximations of the maximum weighted cut problem through derandomization. FST&TCS 9: 1989, Bangalore, India, LNCS 405, 20-29.

[Rag] P. Raghavan. Probabilistic construction of deterministic algorithms: approximating packing integer programs. JCSS 37:4, Oct. 1988, 130-143.

[Sp] J. Spencer. Ten Lectures on the Probabilistic Method. SIAM, Philadelphia, 1987.