A Fast Derandomization Scheme and Its Applications

Yijie Han

Department of Computer Science
University of Kentucky
Lexington, KY 40506

Abstract

We present a fast derandomization scheme for the PROFIT/COST problem. Through the applications of this scheme we show the time complexity of $O(\log^2 n \log\log n)$ for the $\Delta + 1$ vertex coloring problem using $O((m+n)/\log\log n)$ processors on the CREW PRAM, the time complexity of $O(\log^{2.5} n)$ for the maximal independent set problem using $O((m+n)/\log^{1.5} n)$ processors on the CREW PRAM and the time complexity of $O(\log^{2.5} n)$ for the maximal matching problem using $O((m+n)/\log^{0.5} n)$ processors on the EREW PRAM.

Keywords: Derandomization, parallel algorithms, graph algorithms, graph coloring, maximal independent set, maximal matching.

Abbreviated titles: Derandomization.

AMS subject classifications: 05C70, 05C85, 68Q20, 68Q22, 68Q25, 68R10.

1 Introduction

Recent progress in derandomization has resulted in several efficient sequential and parallel algorithms [ABI][BR][BRS][H2][HI][KW][L1][L2][L3][MNN][PSZ][Rag][Sp]. The essence of the technique of derandomization lies in the design of a sample space easy to search, in the probabilistic analysis showing that the expectation of a desired random variable is no less than demanded, and in the search technique which ultimately returns a good sample point. Raghavan[Rag] and Spencer[Sp] showed how to search an exponential size sample space to obtain polynomial time sequential algorithms. Their technique cannot be applied directly to obtain efficient parallel algorithms through derandomization. Alon *et al.*[ABI], Karp and Wigderson[KW] and Luby[L1][L2][L3] developed schemes using $O(n)$ random variables with limited independence on a small sample space, and thus obtained efficient parallel algorithms through derandomization. Berger and Rompel[BR] and Motwani *et al.*[MNN] presented novel designs in which $(\log^c n)$-wise independent random variables are used in randomized algorithms and then $\mathcal{NC}$[Co] algorithms are obtained through the derandomization of these randomized algorithms.

To obtain efficient parallel algorithms, *i.e.* algorithms using no more than a linear number of processors and running in polylog time, Luby[L2][L3] outlined an elegant framework in which pairwise independent random variables are designed on a sample space that facilitates binary search. His framework[L2][L3] consists of a derandomization scheme for the bit pairs PROFIT/COST problem and the general pairs PROFIT/COST problem, and applications of the scheme to three problems: the $\Delta+1$ vertex coloring problem, the maximal independent set problem and the maximal matching problem. By applying his derandomization scheme, he obtained a linear processor CREW(Concurrent Read Exclusive Write) algorithm for the $\Delta+1$ vertex coloring problem with time complexity $O(\log^3 n \log\log n)$. Although he put the three problems in a very nice setting, his derandomization scheme is not efficient enough to improve on parallel algorithms for the maximal independent set problem and the maximal matching problem obtained through ad hoc designs[GS2][IS]. To illustrate his ideas, Luby gave linear processor algorithms for the maximal independent set problem and the maximal matching problem with time complexity $O(\log^5 n)$ through derandomization[L3].

Recently Han and Igarashi[HI] gave a fast derandomization scheme for the bit pairs PROFIT/COST problem. Their scheme yields a fast CREW parallel algorithm for the bit pairs PROFIT/COST problem with time complexity $O(\log n)$ using a linear number of processors. Han then showed[H2] how to obtain an EREW(Exclusive Read Exclusive Write) algorithm with the same time and processor complexities. Their result improves the time complexity of Luby's $\Delta+1$ vertex coloring algorithm to $O(\log^3 n)$. The most interesting feature in Han and Igarashi's scheme[H2][HI] is the design of a sample space which allows redundancy and mutual independence to be exploited.

In this paper we give a new scheme to speed up the derandomization process of the general pairs PROFIT/COST problem. This scheme allows several bit pairs PROFIT/COST problems in one general pairs PROFIT/COST problem to be solved in one pass. We note that our scheme cannot be constructed efficiently under the setting of previous derandomization schemes[L2][L3] because it would require more than a linear number of processors. The power of our derandomization scheme allows us to improve the time complexity for the $\Delta+1$ vertex coloring problem and to obtain new efficient parallel algorithms for the maximal independent set problem and the maximal matching problem.

A substantial amount of effort has been put into the search for efficient parallel algorithms for these three problems. There are important special cases where optimal parallel algorithms are known. Han[H1] has an optimal parallel algorithm for these three problems for linked lists and Hagerup *et al.* [HCD] have an optimal parallel algorithm for the 5-coloring of planar graphs which implies an optimal parallel algorithm for the maximal independent set problem for planar graphs. Significant progress has also been made on the three problems for graphs [ABI][GS1][GS2][HI][IS][KW][L1][L2][L3]. In this paper we only study these three problems on graphs.

Luby obtained through derandomization a CREW algorithm for the $\Delta+1$ vertex coloring problem with time complexity $O(\log^3 n \log\log n)$ using a linear number of processors. Han and Igarashi's work[HI] improves the time complexity for the $\Delta+1$ vertex coloring problem to $O(\log^3 n)$. In this paper we improve the time complexity for the $\Delta+1$ vertex coloring problem to $O(\log^2 n \log\log n)$ using $O((m+n)/\log\log n)$ processors on the CREW PRAM[FW]. For the maximal independent set problem, the first $\mathcal{NC}$ algorithm was obtained by Karp and Wigderson[KW], which has time complexity $O(\log^4 n)$ using $O(n^3/\log^3 n)$ processors on the EREW PRAM. Their result has since been improved to time $O(\log^2 n)$ using $O(mn^2)$ processors on the EREW PRAM by Luby[L1] and to time $O(\log^3 n)$ using $O((m+n)/\log n)$ processors on the EREW PRAM by Goldberg and Spencer[GS1][GS2]. In this paper we obtain an EREW algorithm with time complexity $O(\log^{2.5} n)$ using $O((m+n)/\log^{0.5} n)$ processors. We are able to achieve the same time complexity using $O((m+n)/\log^{1.5} n)$ processors on the CREW PRAM. We also obtain a CREW algorithm with time complexity $O(\log^2 n)$ using $O(n^{2.376})$ processors. For the maximal matching problem, Israeli and Shiloach's CRCW(Concurrent Read Concurrent Write) algorithm[IS] has time complexity $O(\log^3 n)$ using $O(m+n)$ processors. In this paper we give an EREW algorithm with time complexity $O(\log^{2.5} n)$ using $O((m+n)/\log^{0.5} n)$ processors.

Since our algorithms are obtained through derandomization, they are derived at a loss of efficiency from the original randomized algorithms.

Our approach to the three problems follows that of Luby's[L2][L3]. Our results are obtained through the novel design of sample spaces which follows Han and Igarashi's work[H2][HI], the formulation of our fast derandomization process, and the adaptive applications of the fast derandomization techniques to the three problems.

We have outlined here only the main results achieved. There are ramifications of our results which we will mention in the remaining sections of the paper.

2 Preliminaries

The *bit pairs PROFIT/COST* (BPC) and the *general pairs PROFIT/COST* (GPC) problems as formulated by Luby[L2] can be described as follows.

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 $B(\vec{x}) = \sum_{i,j} f_{i,j}(x_i, x_j)$, where $f_{i,j}$ is defined as a function $\{0,1\}^q \times \{0,1\}^q \rightarrow \mathcal{R}$. The general pairs PROFIT/COST problem is to find a good point $\vec{y}$ such that $B(\vec{y}) \geq E[B(\vec{x})]$. B is called the general pairs BENEFIT function and $f_{i,j}$'s are called the general pairs PROFIT/COST functions. When $q = 1$ the GPC problem is called a bit pairs PROFIT/COST problem and $f_{i,j}$'s are called the bit pairs PROFIT/COST functions.

The size m of the problem is the number of nontrivial PROFIT/COST functions present in the input.

Let $G = (V, E)$ be a graph with $|V| = n$ and $|E| = m$. The degree of $v \in V$ is denoted by $d(v)$. Let $\Delta = \max\{d(v) | v \in V\}$. The output of the $\Delta + 1$ vertex coloring is $color(v) \in \{1, ..., \Delta + 1\}$ for all $v \in V$ such that if $(i, j) \in E$ then $color(i) \neq color(j)$. $I \subseteq V$ is an independent set if for $v, w \in I, v \neq w, (v, w) \notin E$. I is a maximal independent set if I is not a proper subset of any other independent set. $M \subseteq E$ is a matching set if no two edges in M have a common vertex. M is maximal if it is not a proper subset of any other matching set.

Han and Igarashi[HI] have formulated the BPC problem as a tree contraction problem[MR]. Without loss of generality assume that n is a power of 2. n 0/1-valued uniformly distributed mutually independent random variables r_i, $0 \le i < n$, are used. A *random variable tree* T is built for $\vec{x}$. T 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 variables x_i, $0 \le i < n$, are associated with n leaves of T and the n random variables r_i, $0 \le i < n$, 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, $0 \le i < n$, are chosen randomly as follows. Let $\vec{r} = < r_i : i = 0, ..., n-1 >$ and let $r_{i_0}, r_{i_1}, ..., r_{i_{\log n}}$ be the random variables on the path from leaf i to the root of T. Random variable x_i is defined to be $x_i(\vec{r}) = (\sum_{j=0}^{\log n - 1} i_j \cdot r_{i_j} + r_{i_{\log n}}) \bmod 2$, where i_j is the j-th bit of i starting with the least significant bit. It can be verified[H2] that random variables x_i, $0 \le i < n$, are uniformly distributed mutually independent random variables.

Due to the linearity of expectation and pairwise independence of random variables in $\vec{x}$, $E[B(\vec{x})] = \sum_{i,j} E[f_{i,j}(x_i, x_j)] = \sum_{i,j} E[f_{i,j}(x_i(\vec{r}), x_j(\vec{r}))] = E[B(\vec{x}\,(\vec{r}))]$. The problem now is to find a sample point $\vec{r}$ such that $B(\vec{r}) \ge E[B] = \frac{1}{4}\sum_{i,j}(f_{i,j}(0,0) + f_{i,j}(0,1) + f_{i,j}(1,0) + f_{i,j}(1,1))$.

Han and Igarashi's algorithm[HI] fixes 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 (level 0) and going upward on the tree T until level $\log n$. Since there are $\log n + 1$ interior levels in T all random variables will be fixed in $\log n + 1$ steps.

Let random variable r_i at level 0 be the parent of the random variables x_i and $x_{i\#0}$ in the random variable tree, where $i\#j$ is a number obtained by complementing the j-th bit of i. r_i will be fixed as follows. Compute $f_0 = E[f_{i,i\#0} + f_{i\#0,i} | r_i = 0] = (f_{i,i\#0}(0,0) + f_{i,i\#0}(1,1) + f_{i\#0,i}(0,0) + f_{i\#0,i}(1,1))/2$ and $f_1 = E[f_{i,i\#0} + f_{i\#0,i} | r_i = 1] = (f_{i,i\#0}(0,1) + f_{i,i\#0}(1,0) + f_{i\#0,i}(0,1) + f_{i\#0,i}(1,0))/2$. If $f_0 \ge f_1$ then set r_i to 0 else set r_i to 1. All random variables at level 0 will be fixed in parallel in constant time using n processors. This results in a smaller space with higher expectation for B. Therefore this smaller space contains a good point.

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, $0 \le i < n$, can be reduced to $n/2$ random variables. PROFIT/COST 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 as B but has only $n/2$ variables. As we stated above, $E[B^{(1)}] \ge E[B]$.

What we have explained above is the first step of the algorithm in [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 $\{x_0^{(k)}, x_1^{(k)}, ..., x_{n/2^k-1}^{(k)}\}$, and functions $f_{i,j}$, $i,j \in \{0,1,...,n-1\}$, have been combined into $f_{i,j}^{(k)}$, $i,j \in \{0,1,...,n/2^k-1\}$.

After $\log n$ steps $B^{(\log n)} = f_{0,0}^{(\log n)}(x_0^{(\log n)}, x_0^{(\log n)})$. The bit at the root of the random variable tree is now set to 0 if $f_{0,0}^{(\log n)}(0,0) \geq f_{0,0}^{(\log n)}(1,1)$, 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.

Let $n = 2^k$ and A be an $n \times n$ array. Elements $A[i,j]$, $A[i,j\#0]$, $A[i\#0,j]$, $A[i\#0,j\#0]$ form a *gang* which is denoted by $g_A[\lfloor i/2 \rfloor, \lfloor j/2 \rfloor]$. All gangs in A form array g_A.

When visualized on a two dimensional array A (as shown in Fig. 1), a stage of Han and Igarashi's algorithm can be interpreted as follows. Let function $f_{i,j}$ be stored at $A[i,j]$. Setting the random variables at level 0 of the random variable tree is done by examining the PROFIT/COST functions in the diagonal gang of A. Function $f_{i,j}$ then gets the bit setting information from $g_A[\lfloor i/2 \rfloor, \lfloor i/2 \rfloor]$ and $g_A[\lfloor j/2 \rfloor, \lfloor j/2 \rfloor]$ to determine how it is to be combined with other functions in $g_A[\lfloor i/2 \rfloor, \lfloor j/2 \rfloor]$.

A *derandomization tree* D can be built which reflects the way the BPC functions are combined. D is of the following form. The input BPC functions are stored at the leaves, $f_{i,j}$ is stored in $A_0[i,j]$. A node $A_l[i,j]$ at level $l > 0$ is defined if there exist input functions in the range $A_0[u,v]$, $i * 2^l \leq u < (i+1) * 2^l$, $j * 2^l \leq v < (j+1) * 2^l$. A derandomization tree is shown in Fig. 2.

Tree D can be constructed[H2][HI] by first sorting the input into the file-major indexing and then building the tree bottom-up. The derandomization tree helps to reduce the space requirement for the BPC problem. Han and Igarashi's algorithm[HI] has time complexity $O(\log n)$ using a linear number of processors.

The algorithm given by Han and Igarashi[HI] is a CREW algorithm. Recently, Han[H2] has given an EREW algorithm for the BPC problem with time complexity $O(\log n)$ using a linear number of processors.

We now discuss some variations of the above algorithm to be used in sections 5 and 6.

In some applications the BPC functions cannot be combined in order to obtain an efficient algorithm. If the functions are not combined then there could be several BPC functions $f_1(x_{i_1}, x_{j_1}), f_2(x_{i_2}, x_{j_2}), ..., f_l(x_{i_k}, x_{j_k})$ associated with an interior node r in the random variable tree, where x_{i_t}, x_{j_t} are leaves in the subtree rooted at r. If r is not the root of the whole random variable tree then one of x_{i_t}, x_{j_t} is in the left subtree of r and the other in the right subtree of r, $1 \leq t \leq k$.

Let x_i be a leaf of a random variable subtree T. Let the random variables on the path from x_i to the root r of T be set to $a_0, a_1, ..., a_l$. We define $\Psi(x_i, r) = \sum_{j=0}^{l}(a_j \cdot i_j) \bmod 2$. This function resembles the Ψ function defined by Luby[L2][L3].

In fixing r we tentatively set r to 0 and 1 respectively and evaluate $f_t(\Psi(x_{i_t}, r), \Psi(x_{j_t}, r)) + f_t(\Psi(x_{i_t}, r) \oplus$

$1, \Psi(x_{j_t}, r) \oplus 1)$, $1 \le t \le k$, where $\oplus$ is the exclusive-or function. We then get the sum of these functions and compare the sum for $r = 0$ with the sum for $r = 1$ to decide whether r should be set to 0 or 1. It takes $O(\log n)$ time to get the sum because there are at most $O(n^2)$ functions.

We note that $\Psi(x_i, r)$ can be evaluated progressively as the derandomization process proceeds, *i.e.* $\Psi(x_i, r) = (\Psi(x_i, r') + tr) \bmod 2$, where r is the parent of r' and t is the bit of i corresponding to r.

Thus if the functions are not combined in the derandomization process a BPC problem requires $O(\log^2 n)$ time to solve.

In our applications we also use a combination of Luby's technique[L2] and Han and Igarashi's technique[H2][HI] for solving a BPC problem. The random variable tree used in Luby's algorithm degenerates to a chain of length $\log n + 1$ plus n leaves. Therefore there are $\log n + 1$ random variables $r_0, r_1, ..., r_{\log n}$ associated with the interior nodes in the tree. x_i is chosen randomly by the formula $x_i = (\sum_{j=0}^{\log n - 1} i_j \cdot r_j + r_{\log n}) \bmod 2$. It can be shown[L2] that x_i's are pairwise independent random variables. In Luby's algorithm the random variables in the random variable tree are also fixed one level at a time. His algorithm takes $O(\log n)$ time to fix one level resulting in time complexity $O(\log^2 n)$. We stress that the random variable tree used in Luby's algorithm has only $\log n + 1$ random variables. Thus the sample space contains only $2n$ sample points, while the sample space used in Han and Igarashi's algorithm[H2][HI] has 2^n sample points.

Combining Luby's technique and Han and Igarashi's technique[H2][HI], we could solve a BPC problem by using a random variable tree T as shown in Fig. 3c. T has $S = \lceil (\log n + 1)/a \rceil$ blocks, where a is a parameter. Block s contains levels as to $a(s+1) - 1$ of T. Block 0 has $C_0 = \lceil n/2^a \rceil$ chains. Block s has $C_s = \lceil C_{s-1}/2^a \rceil$ chains. Each chain in block s has length a running from level as to level $a(s+1) - 1$. A node at level as in a chain (except the one in the last chain) has 2^a children at level $as - 1$. Block $S - 1$ has only one chain of length $\log n + 1 - a(S - 1)$. There is a random bit r at each interior node of T and random variable x_i is associated with the i-th leaf of T. x_i is chosen randomly as $x_i(\vec{r}) = (\sum_{j=0}^{\log n - 1} i_j \cdot r_{i_j} + r_{i_{\log n}}) \bmod 2$, where $r_{i_0}, r_{i_1}, ..., r_{i_{\log n}}$ are the random variables on the path from leaf i to the root of T. It is straightforward to show that the x_i's are uniformly distributed pairwise independent random variables.

Different random variable trees are shown in Fig. 3.

3 A Scheme for the General Pairs PROFIT/COST Problem

In this section we present a scheme to speed up the derandomization process for the GPC problem.

In [L2][L3] Luby presented the following derandomization scheme for solving the GPC problem.

Let $\vec{y} =< y_i \in \{0,1\}^p : i = 0, 1, ..., n-1 >$. Let $\vec{x_u}$, $p \le u < q$, be totally independent random bit strings, each of length n. Let $\vec{z}$ be a vector of n bits. We write the BENEFIT function $B(x_0, x_1, ..., x_{n-1})$, where each x_i is a variable containing q bits, as $B(\vec{x_{q-1}} \cdots \vec{x_{p+1}} \vec{x_p} \vec{y})$, where $\vec{y}$ contains the least significant p bits of all variables and $\vec{x_i}$ contains the i-th bits of all variables. Define $TB(\vec{y}) = E[B(\vec{x_{q-1}} \cdots \vec{x_{p+1}}\vec{x_p}\vec{y})]$. Then $E[TB(\vec{x_p}\vec{y})] = E[E[B(\vec{x_{q-1}} \cdots \vec{x_{p+1}}\vec{z}\vec{y})| \vec{z} =\vec{x_p}]] = E[B(\vec{x_{q-1}} \cdots \vec{x_p}\vec{y})] = TB(\vec{y})$. That is, $TB(\vec{y})$ can also be obtained by first computing $TB(\vec{x_p}\vec{y})$, and $TB(\vec{y}) = E[TB(\vec{x_p}\vec{y})] = \sum_{\vec{z}} TB(\vec{z}\vec{y})Pr(\vec{x_p}=\vec{z} \mid \vec{x_{p-1}} \cdots \vec{x_0}=\vec{y})$. Thus there exists a $\vec{z}$ such that $TB(\vec{z}\vec{y}) \geq TB(\vec{y})$. After $TB(\vec{x_p}\vec{y})$ are evaluated for all values of $\vec{x_p}$, the problem of finding such a $\vec{z}$ is a BPC problem. Because in the GPC problem function B is the sum of GPC functions, each depending on at most two variables, pairwise independent random variables can be used for bits in each $\vec{x_u}$, $p+1 \leq u < q$. Luby's algorithm for the GPC problem then uses his algorithm for the BPC problem 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 solution[L2][L3] to the GPC problem can be interpreted as follows: it solves the GPC problem by solving q BPC problems, one for each $\vec{x_u}$. These BPC problems are solved sequentially. After the BPC problems for $\vec{x_u}$, $0 \leq u < v$, are solved. BPC functions $f_{i_v,j_v}(x_{i_v}, x_{j_v})$ are evaluated based on the setting of bits x_{i_u}, x_{j_u}, $0 \leq u < v$. Suppose x_{i_u} is set to y_{i_u} and x_{j_u} is set to y_{j_u}, $0 \leq u < v$. $f_{i_v,j_v}(x_{i_v}, x_{j_v})$ is evaluated as $f_{i_v,j_v}(y_{i_v}, y_{j_v}) = E[f_{i,j}(x_i, x_j)|x_{i_0} = y_{i_0}, x_{j_0} = y_{j_0}, ..., x_{i_{v-1}} = y_{i_{v-1}}, x_{j_{v-1}} = y_{j_{v-1}}, x_{i_v} = y_{i_v}, x_{j_v} = y_{j_v}]$, $y_{i_v}, y_{j_v} = 0, 1$. After BPC functions $f_{i_v,j_v}(x_{i_v}, x_{j_v})$ have been obtained and stored in a table, the BPC algorithm is invoked to fix $\vec{x_v}$.

If we are to solve several BPC problems in a GPC problem simultaneously we must have BPC functions $f_{i_v,j_v}(x_{i_v}, x_{j_v})$ before the setting of the bits x_{i_u}, x_{j_u}, $0 \leq u < v$. Since there are a total of $2v$ bits we could try out all possible 4^v bit patterns. For each bit pattern we have a distinct function $f_{i_v,j_v}(x_{i_v}, x_{j_v})$. If $q = O(\log n)$ we need only a polynomial number of processors to work on all these functions.

If we use Luby's random variable tree for each BPC problem then there are $\log n + 1$ random variables and $2n$ sample points for each BPC problem. Thus if we try to solve for $\vec{x_v}$ before $\vec{x_u}$, $0 \leq u < v$, are solved, we have to take care of $(2n)^v$ possible situations. Apparently more than a polynomial number of processors are needed if v is not a constant. So how about using Han and Igarashi's random variable tree[H2][HI]? There are now n random variables and 2^n sample points for each BPC problem. The situation seems to be even more difficult to deal with. However, by close examination we find out that instead we could reduce the number of processors by using Han and Igarashi's random variable tree.

We now present a scheme which allows several $\vec{x_u}$'s to be fixed in one pass using Han and Igarashi's random variable tree for each BPC problem.

First we give a sketch of our approach. The incompleteness of the description in this paragraph will be elaborated on in the rest of this section. 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 section. Note that BPC functions

$f_{i_v,j_v}(x_{i_v}, x_{j_v})$ depend 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 $\vec{x_v}$, the $\vec{x_u}$, $0 \leq u < v$, 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}, $0 \leq j < u$. We use F_u to denote the random variable forest for P_u. We 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 $l+c-1$, $0 \leq c \leq v$, in F_{v-c} have already been fixed. We 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 in which we 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 the function values stored in a table. However, because $\vec{x_u}$, $0 \leq u < v$, have not been fixed yet, we have to try out all possible cases. There are a total of 4^v patterns for bits x_{i_u}, x_{j_u}, $0 \leq u < v$. We use 4^v BPC functions for each pair (i,j). We use $f_{i_v,j_v}(x_{i_v}, x_{j_v})(y_{v-1}y_{v-2}\cdots y_0, z_{v-1}z_{v-2}\cdots z_0)$ to 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), $0 \leq l \leq \log n$, a *bit pipeline tree* is built (Fig. 4) which is a complete binary tree of height $2k$. Nodes at even depth from the root in a bit pipeline tree are selectors, and 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. One random variable corresponds to each selector. Let random variable r correspond 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. 4 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 1: 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. $\square$

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 signals select 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 1 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 the 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 2: $LS^l_{w,d} = RS^l_{w,d} = \{0,1\}^d$.

Proof: By induction. Assume that the lemma 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. □

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 2 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 a superscript 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 3: $|S| = 4^d$.

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 the 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}. □

Lemma 3 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 *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 1: The GPC problem can be solved on the CREW PRAM in time $O((q/k+1)(\log n+k+\tau))$ with $O(4^k m)$ processors, where τ is the time for a single processor to evaluate a BPC function $f_{i_d,j_d}(x_{i_d}, x_{j_d})(\alpha,\beta)$.

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. To solve the u-th BPC problem is to evaluate $P_u(\vec{z}) = E[B(\vec{x_{q-1}} \cdots \vec{x_{u+1}}\,\vec{z}\,\vec{y})]$, $\vec{z} \in \{0,1\}^n$, where $\vec{y} =< y_i \in \{0,1\}^u : i=0,1,...,n-1 >$ is fixed. We then view $\vec{z}$ as a random variable uniformly distributed on $\{0,1\}^n$ and find $\vec{z'}$ such that $P_u(\vec{z'}) \geq E[P_u(\vec{z})]$. If we have a huge number of processors we could solve all BPC problems in parallel by solving each P_u with all possible $\vec{y}$'s. Such an algorithm is apparently correct. In our scheme $P_u(\vec{z})$ is evaluated by evaluating $E[f_{i,j}(\alpha\beta, \alpha'\beta')]$, $\alpha,\alpha' \in \{0,1\}$, $\beta,\beta' \in \{0,1\}^u$. This is guaranteed to be correct by linearity of expectation. We used a pipeline to solve the P_u's. Thus our algorithm is still correct while the number of processors needed is drastically reduced.

With $O(4^k m)$ processors k $\vec{x_u}$'s are fixed in one pass. Each pass takes $O(\log n + k + \tau)$ time, τ for evaluating BPC functions (*i.e.* setting up the function tables for the BPC problems) and $O(\log n + k)$ time for fixing all random bits on the random variable trees. The time complexity for solving the GPC problem is $O((q/k+1)(\log n + k + \tau))$. $\Box$

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 better understand 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 $t$ $(0 \leq t < u)$: Wait for the pipeline to be filled.

    Step $u+t$ $(0 \leq t < \log n)$:
        (* In this step we will 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 $T_{u-1}^{(t-1)}[i0][j]$, and $T_u^{(t-1)}[i1][j'0]$ 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 $T_u^{(t-1)}[i1][j'r]$ 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 $T_u^{(t-1)}[i1][j'\overline{r}]$ 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 $T_u^{(t)}[i][j1]$ the children of $T_{u-1}^{(t)}[i][j]$ in the 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 will 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 $T_u^{(\log n-1)}[0][jr]$ the child of $T_u^{(\log n)}[0][jr]$ in the random variable tree;

        make $T_u^{(\log n)}[0][jr]$ 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 a dynamic programming technique. These are some of the essential elements of our scheme.

An example of the execution of procedure RV-Tree is shown in Fig. 5.

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 $f^{(l)}_{u_k,v_k}(x_{u_k},\ x_{v_k})$, 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 $f^{(l)}_{u_k,u\#0_k}(x_{u_k}, x_{u\#0_k})(\alpha,\beta)$ to acquire the bit pattern α' which satisfies $(\alpha',\beta) \in S^l_{u,k}$. Function $f^{(l)}_{u_k,v_k}(x_{u_k}, x_{v_k})(\alpha,\beta)$ can then obtain the bit pattern α' from $f^{(l)}_{u_k,u\#0_k}(x_{u_k}, x_{u\#0_k})(\alpha,\beta)$ by the pipeline scheme described in [H2].

Suppose we are to solve P_u, $0 \leq u \leq k$, in one pass. We solve 4^k copies of P_{k-1}; one copy corresponds to one conditional bit pattern in P_k. $f^{(l)}_{u_k,v_k}(x_{u_k}, x_{v_k})(\alpha,\beta)$ 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, $0 \leq u \leq k$, in one pass, where c is a suitable constant. Note also that it takes $O(k^2)$ time to make needed copies.

Theorem 2: The GPC 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 a single processor to evaluate a BPC function $f_{u_d,v_d}(x_{u_d}, x_{v_d})(\alpha,\beta)$. $\square$

4 $\Delta + 1$ Vertex Coloring

We apply our scheme to Luby's formulation of the $\Delta + 1$ vertex coloring problem[L2][L3]. First we adapt his formulation and then apply our fast derandomization scheme to obtain a faster algorithm. Luby showed[L2][L3] that after solving a GPC problem a constant fraction of the vertices can be deleted. The main change now is to show that after solving a GPC problem a constant fraction of the edges can be deleted. We follow the notations and definitions as given by Luby[L2][L3].

Let $G = (V, E)$ be the graph we are to color. Let $adj(i)$ be the set of vertices which are adjacent to vertex i, and $d(i)$ be the degree of vertex i. Let $avail_i$ be the set of colors which can be used to color vertex i and let $Navail_i = |avail_i|$. Let k_i be such that $2^{k_i-1} < 4Navail_i \leq 2^{k_i}$ and let $Nlist_i = 2^{k_i}$. Let $list_i[0, ..., Nlist_i - 1]$ be an array such that the first $Navail_i$ entries in $list_i$ are the elements of $avail_i$ in sorted order and the remaining entries in $list_i$ have value Λ. Let $q = \max\{k_i | i \in V\}$. Let $\vec{x} = < x_i \in \{0,1\}^q, i \in V >$. For $i \in V$, let $list_i(x_i)$ be the entry in $list_i$ indexed by the first k_i bits of x_i. Also define the following functions.

For all $i \in V$, let

$$Y_i(x_i) = \begin{cases} 1 \text{ if } list_i(x_i) \in avail_i \\ 0 \text{ if } list_i(x_i) = \Lambda \end{cases}$$

For all $(i,j) \in E$, let

$$Y_{i,j}(x_i, x_j) = \begin{cases} -1 \text{ if } list_i(x_i) = list_j(x_j) \neq \Lambda \\ 0 \text{ otherwise} \end{cases}$$

The BENEFIT function B is defined as

$$B(\vec{x}) = \sum_{i \in V} \frac{d(i)}{2} \left(Y_i(x_i) + \sum_{j \in adj(i)} Y_{i,j}(x_i, x_j) \right)$$

Function B sets a lower bound on the number of edges deleted[L2][L3] should the vertex i be tentatively assigned color $list_i(x_i)$. We will not repeat the definitions of the auxiliary functions TY_i and $TY_{i,j}(x_i, x_j)$ as their definitions can be found in [L2][L3]. The auxiliary function TB is now defined as

$$TB\ (\vec{x}) = \sum_{i \in V} \frac{d(i)}{2} \left(TY_i\ (x_i) + \sum_{j \in adj(i)} TY_{i,j}\ (x_i, x_j) \right)$$

Following Luby's proof[L2][L3] we have $TY_i(\Lambda) \geq 1/8$, $TY_{i,j}(\Lambda, \Lambda) \geq -1/16$, therefore,

Lemma 4: $TB\ (\vec{\Lambda}) \geq |E|/16$. $\Box$

Thus by solving a GPC problem[1] we are guaranteed to eliminate a constant fraction of the edges.

Let $c^k m$ be the number of processors needed to compute k BPC problems in a GPC problem in one pass. There will be $O(\log n)$ stages in the modified algorithm. Each stage contains a constant number of GPC problems and reduces the number of edges so that there will be no more than a $1/c$ fraction of the edges left. Therefore during stage i there will be e edges in the remaining graph and $c^i e$ processors available. Because each stage has $O(\log n)$ BPC problems, the time complexity for stage i is $O(\log^2 n/i)$. Thus the time complexity of the whole algorithm becomes $O(\sum_{i=1}^{O(\log n)} \log^2 n/i) = O(\log^2 n \log\log n)$.

The number of processors used in the algorithm can be reduced to $O((m+n)/\log\log n)$. We examine the first $O(\log\log\log n)$ stages. In stage i we can have $c^i/\log\log n$ processors for each edge under each conditional bit pattern. Therefore the tables for the BPC functions in stage i can be computed in time $O(\log^2 n \log\log n/c^i)$ and the overall time for table construction for the whole algorithm is $O(\log^2 n \log\log n)$. The calculation for the time for constructing the derandomization trees is similar and can be shown to be $O(\log^2 n \log\log n)$ with $O((m+n)/\log\log n)$ processors. In the first $O(\log\log\log n)$ stages our GPC algorithm will be invoked with $k = 1$. The time complexity for these stages is

$$O\left(\sum_{i=0}^{O(\log\log\log n)} \frac{\log^2 n \log\log n}{c^i} \right) = O(\log^2 n \log\log n).$$

The remaining stages take $O(\log^2 n \log\log n)$ time by the analysis in the last paragraph.

[1]The problem formulated[L2][L3] resembles a GPC problem. It is not a GPC problem in the strict sense. For our purpose we may view it as a GPC problem because our GPC algorithm applies.

Theorem 3: There is a CREW PRAM algorithm for the $\Delta+1$ vertex coloring problem with time complexity $O(\log^2 n \log\log n)$ using $O((m+n)/\log\log n)$ processors. $\Box$

We also have,

Theorem 4: $O(mn^{\epsilon})$ processors are sufficient to solve the $\Delta+1$ vertex coloring problem in time $O(\log^2 n)$ on the CREW PRAM, where $\epsilon > 0$ is an arbitrary constant.

Proof: This is because one GPC problem can now be solved in $O(\log n)$ time. $\Box$

5 Maximal Independent Set

Let $G=(V,E)$ be an undirected graph. For $W \subseteq V$ let $N(W) = \{i \in V \mid \exists\, j \in W, (i,j) \in E\}$. Known parallel algorithms[ABI][KW][GS1][GS2][L1][L3] for computing a maximal independent set have the following form.

```
**Procedure** General-Independent
**begin**
    $I := \phi$;
    $V' := V$;
    **while** $V' \neq \phi$ **do**
        **begin**
            Find an independent set $I' \subseteq V'$;
            $I := I \cup I'$;
            $V' := V' - (I' \cup N(I'))$;
        **end**
**end**
```

Luby's work[L3] formulated each iteration of the while-loop in General-Independent as a GPC problem. We now adapt Luby's formulation[L1][L3].

Let k_i be such that $2^{k_i-1} < 4d(i) \leq 2^{k_i}$. Let $q = \max\{k_i | i \in V\}$. Let $\vec{x} =< x_i \in \{0,1\}^q, i \in V\}$. The length $|x_i|$ of x_i is defined to be k_i. Define[2]

[2]In Luby's formulation[L3] $Y_i(x_i)$ is zero unless the first $|x_i|$ bits of x_i are 1's. In order to be consistent with the notations in our algorithm we let $Y_i(x_i)$ be zero unless the first $|x_i|$ bits of x_i are 0's.

$$
\begin{aligned}
Y_i(x_i) &= \begin{cases} 1 & \text{if } x_i(|x_i|-1)\cdots x_i(0) = 0^{|x_i|} \\ 0 & \text{otherwise} \end{cases} \\
Y_{i,j}(x_i,x_j) &= -Y_i(x_i)Y_j(x_j) \\
B(\vec{x}) &= \sum_{i\in V}\frac{d(i)}{2}\sum_{j\in adj(i)}\left(Y_j(x_j) + \sum_{k\in adj(j),d(k)\geq d(j)} Y_{j,k}(x_j,x_k) + \sum_{k\in adj(i)-\{j\}} Y_{j,k}(x_j,x_k)\right)
\end{aligned}
$$

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

Function B sets a lower bound on the number of edges deleted from the graph[L1][L3] should vertex i be tentatively labeled as an independent vertex if $x_i = (0\cup 1)^{q-|x_i|}0^{|x_i|}$. The following lemma was proven in [L1] (Theorem 1 in [L1]).

Lemma 5[L1]: $E[B] \geq |E|/c$ for a constant $c > 0$. □

Function B can be written as

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

where

$$
f_i(x_i) = (\sum_{j\in adj(i)}\frac{d(j)}{2})Y(x_i)
$$

and

$$
f_{i,j}(x_i,x_j) = \delta(i,j)(\sum_{k\in adj(i)}\frac{d(k)}{2})Y_{i,j}(x_i,x_j) + (\sum_{k\in V \text{ and } i,j\in adj(k)}\frac{d(k)}{2})Y_{i,j}(x_i,x_j)
$$

$$
\delta(i,j) = \begin{cases} 1 & \text{if } (i,j)\in E \text{ and } d(j)\geq d(i) \\ 0 & \text{otherwise} \end{cases}
$$

Thus each execution of a GPC procedure eliminates a constant fraction of the edges from the graph. It takes $O(M(n))$ (which is $O(n^{2.376}$ currently [CW]) processors to compute a matrix multiplication in time

$O(\log n)$ to arrive at the GPC functions f_i's and $f_{i,j}$'s because of the term $\sum_{i \in V} \frac{d(i)}{2} \sum_{j,k \in adj(i), j \neq k} Y_{j,k}(x_j, x_k)$ in function B. We organize our algorithm for the maximal independent set problem into $O(\log n)$ stages such that in stage i the graph has no more than $|E|/c^i$ vertices and a constant number of GPC problems will be solved in stage i. By Theorem 1 we achieve time complexity $O(\log^2 n)$.

Theorem 5: There is a CREW PRAM algorithm for the maximal independent set problem with time complexity $O(\log^2 n)$ using $O(M(n))$ processors.

Proof: The time and processor complexities for computing matrix multiplication dominate. □

We will give a second algorithm for the maximal independent set problem. We take advantage of the special properties of the GPC functions to reduce the number of processors to $O(m+n)$. We cannot use the derandomization scheme in section 3 directly because it would involve a matrix multiplication as we have seen in the design of our first maximal independent set algorithm. The structure of our second algorithm is complicated. We first give an overview of the algorithm.

Overview of the Second Algorithm

Because we can reduce the number of edges by a constant fraction after solving a GPC problem, a maximal independent set will be computed after $O(\log n)$ GPC problems are solved. Our algorithm has two stages, the initial stage and the speedup stage. The initial stage consists of the first $O(\log^{0.5} n)$ GPC problems. Each GPC problem is solved in $O(\log^2 n)$ time. The time complexity for the initial stage is thus $O(\log^{2.5} n)$. When the first stage finishes the remaining graph has size $O((m+n)/2^{\sqrt{\log n}})$. There are $O(\log n)$ GPC problems in the speedup stage. A GPC problem of size s in the speedup stage is solved in time $O(\log^2 n/\sqrt{k})$ with $O(c^k s \log n)$ processors. Therefore the time complexity of the speedup stage is $O(\sum_{i=O(\sqrt{\log n})}^{O(\log n)} (\log^2 n/\sqrt{i})) = O(\log^{2.5} n)$. The initial stage is mainly to reduce the processor complexity while the speedup stage is mainly to reduce the time complexity.

We used matrix multiplication in our first algorithm because of the term $\sum_{i \in V} \frac{d(i)}{2} \sum_{j,k \in adj(i), j \neq k} Y_{j,k}(x_j, x_k)$ in function B. We shall call this term the vertex cluster term. There is a cluster $C(v) = \{x_w | (v,w) \in E\}$ for each vertex v. Alternatively we may use $O(\sum_{v \in V} d^2(v))$ processors, $d^2(v)$ processors for cluster $C(v)$, to evaluate all GPC functions and to apply our derandomization scheme given in section 3. However, to reduce the number of processors to $O(m+n)$ we have to use a modified version of our derandomization scheme in section 3.

Consider the problem of fixing a random variable r in the random variable tree. We did this in constant time in section 3 (Theorem 1). We now outline how r is fixed in the initial stage. We cannot do it in constant time because the GPC function $f(x,y)$, where x and y are the leaves in the subtree rooted at r, is in fact the sum of several functions scattered in the second term of function B and in several clusters. We will not combine BPC functions in the derandomization process. As we have explained in section 2, setting r requires $O(\log n)$ time because of the summation of function values. (Note that the summation of n items can be done in time $O(n/p + \log n)$ time with p processors.) A BPC problem takes $O(\log^2 n)$ time to solve. We pipeline all BPC problems in a GPC problem and get time complexity $O(\log^2 n)$ for solving a GPC problem.

The functions in B have a special property which we will exploit in our algorithm. Each variable x_i has a length $|x_i| \leq q = O(\log n)$. $Y_{i,j}(x_i, x_j)$ is zero unless the first $|x_i|$ bits of x_i are 0's and the first $|x_j|$ bits of x_j are 0's. When we apply our scheme there is no need to keep BPC functions $Y_{i_u,j_u}(x_{i_u}, x_{j_u})$ for all conditional bit patterns because many of these patterns will yield zero BPC functions. In our algorithm we keep one copy of $Y_{i_u,j_u}(x_{i_u}, x_{j_u})$ with conditional bits set to 0's. This of course helps reduce the number of processors. In particular, the random variable tree for each P_u now requires at most $O(n)$ processors, instead of $O(c^u n)$ processors as we have used in section 3 on the CREW PRAM.

There are $d^2(i)$ BPC functions in cluster $C(i)$ while we can allocate at most $d(i)$ processors in the very first GPC problem because we have at most $O(m + n)$ processors for the GPC problem. What we do is use an *evaluation tree* for each cluster. The evaluation tree $TC(i)$ for cluster $C(i)$ is a "subtree" of the random variable tree. The leaves of $TC(i)$ are the variables in $C(i)$. An interior node of the random variable tree is not present in $TC(i)$ if none of the leaves of the subtree rooted at the interior node is in $C(i)$. When we are fixing r, if r is not in $TC(i)$ then cluster $C(i)$ does not contribute anything. If r is in $TC(i)$ then the contribution of $C(i)$ can be obtained by evaluating the function $f(x, y)$, where x, y are leaves in the evaluation subtree rooted at r and x and y are in different subtrees of r. If r has a leaves in the left subtree and b leaves in the right subtree then the contribution from $TC(i)$ for fixing r is the sum of ab function values. We will give the details of evaluating this sum using a constant number of operations.

Let us summarize the main ideas. We do not combine functions and achieve time $O(\log^2 n)$ for solving a BPC problem; we put all BPC problem in a GPC problem as one batch into a pipeline to get $O(\log^2 n)$ time for solving a GPC problem; we use a special property of functions in B to maintain one copy for each BPC function for only conditional bits of all 0's; we use evaluation trees to take care of the vertex cluster term.

We now sketch the speedup stage. Since we have to solve $O(\log n)$ GPC problems in this stage, we have to reduce the time complexity for a GPC problem to $o(\log^2 n)$ in order to obtain $o(\log^3 n)$ time. We use a modified random variable tree as shown in Fig. 3c in section 2. Such a random variable tree has $S = \lceil (\log n + 1)/a \rceil$ blocks. Each block contains a levels. We fix a block in a step instead of fixing a level in a step. Each step takes $O(\log n)$ time and a BPC problem takes $O(S \log n)$ time. If we have as many processors as we want we could solve all BPC problems in a GPC problem by enumerating all possible cases instead of putting them through a pipeline; *i.e.*, in solving P_u we could guess all possible settings of random variables for P_v, $0 \leq v < u$. We have explained this approach in the proof of Theorem 1 in section 2. In doing so we would achieve time $O(S \log n)$ for solving a GPC problem. In reality we have extra processors, but they are not enough for us to enumerate all possible situations. We therefore put a BPC problems of a GPC problem in a team. All BPC problems in a team are solved by enumeration. Thus they are solved in time $O(S \log n)$. Let b be the number of teams we have. We put all these teams into a pipeline and solve them in time $O((S + b) \log n)$. The approach of the speedup stage can be viewed as that of the initial stage with added parallelism which comes with the help of extra processors.

The Initial Stage

We first show how to solve a GPC problem for function B in time $O(\log^2 n)$ using $O((m+n)\log n)$ processors.

$O(m+n)$ processors will be allocated to each BPC problem. The algorithm for processors working on F_u has the following form.

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

Step $u+t$ $(0 \le t < \log n)$: Fix random variables at level t.

Step $u+\log n$: Fix the only remaining random variable at level $\log n$. Output the good point for $\vec{x_u}$.

We will allow $O(\log n)$ time for each step and $O(\log^2 n)$ time for the whole algorithm. Note that we do not combine functions with respect to the surviving set and therefore use $O(\log n)$ time for a step.

The way T_u is constructed can be described by algorithm MRV-Tree, a modified version of algorithm RV-Tree in section 3. In MRV-Tree we do not enumerate all possible conditional bit patterns. Only the bit pattern of all 0's is kept. Thus a node on a bit pipeline tree may not have both children. A variable x_i only appears in F_u as x_{i_u} with $u < |x_i|$ because the setting of random variables in F_u, $u \ge |x_i|$, is not affected by x_{i_u}.

```
Procedure **MRV-Tree**
begin
  Step $t$ $(0 \le t < u)$: Wait for the pipeline to be filled.

  Step $u$:
    (* In this step we will build $T_u^{(0)}[i][j]$. $0 \le i < n/2$ and $j$ are indices for which $T_u^{(0)}[i][j]$ is not
    empty. At the beginning of this step $T_{u-1}^{(0)}[i][j]$ has already been constructed if it is not empty.
    Random variables $x_i$ have been transmitted to depth $u$ of the bit pipeline tree for level 0. *)

    for each node $T_u^{(0)}[i][j]$
      if $T_u^{(0)}[i][j]$ has received either $x_{2i}$ or $x_{2i+1}$ from $T_{u-1}^{(0)}[i][j/2]$
        (* $x_{2i}$ and $x_{2i+1}$ becomes $x_{2i_u}$ and $x_{2i+1_u}$ in $T_u$. *) then
        begin
          if $T_u^{(0)}[i][j]$ has received $x_{2i}$ then
            make it the left child of $T_u^{(0)}[i][j]$ in the random variable forest for $P_u$;
        if $T_u^{(0)}[i][j]$ has received $x_{2i+1}$ then
            make it the right child of $T_u^{(0)}[i][j]$ in the random variable forest for $P_u$;

        fix random variable $r$ in $T_u^{(0)}[i][j]$;

        make $T_u^{(0)}[i][j]$ a child of $T_{u-1}^{(0)}[i][j/2]$ in the bit pipeline tree;

        if $T_u^{(0)}[i][j]$ has received $x_{2i}$ and $|x_{2i}| \geq u$ then
            transmit $x_{2i}$ to $T_{u+1}^{(0)}[i][j0]$;

        if $T_u^{(0)}[i][j]$ has received $x_{2i+1}$ and $|x_{2i+1}| \geq u$ then
            transmit $x_{i\#0}$ to $T_{u+1}^{(0)}[i][jr]$;
    end
  else (* $T_u^{(0)}[i][j]$ is empty. *);
```

Step $u + t$ $(1 \leq t < \log n)$:

(* In this step we will build $T_u^{(t)}[i][j]$, $0 \leq i < n/2^{t+1}$ and j are indices for which $T_u^{(t)}[i][j]$ is not empty. 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 $T_{u-1}^{(t-1)}[i1][j']$ (they may be empty) be the two children in the random variable subtree rooted at $T_{u-1}^{(t)}[i][j]$. $T_u^{(t-1)}[i0][j0]$ and $T_u^{(t-1)}[i0][j1]$ (they may be empty) are the children of $T_{u-1}^{(t-1)}[i0][j]$, and $T_u^{(t-1)}[i1][j'0]$ and $T_u^{(t-1)}[i1][j'1]$ (they may be empty) 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. *)

```
if $T_u^{(t-1)}[i0][j0]$ is not empty then
    make it the left child of $T_u^{(t)}[i][j0]$ in the random variable forest for $P_u$;
if $T_u^{(t-1)}[i1][j'r]$ is not empty then
    make it the right child of $T_u^{(t)}[i][j0]$ in the random variable forest for $P_u$;

if $T_u^{(t-1)}[i0][j1]$ is not empty then
    make it the left child of $T_u^{(t)}[i][j1]$ in the random variable forest for $P_u$;
if $T_u^{(t-1)}[i1][j'\overline{r}]$ is not empty then
    make it the right child of $T_u^{(t)}[i][j1]$ in the random variable forest for $P_u$;

if $T_u^{(t)}[i][j0]$ is not empty then
    make it the left child of $T_{u-1}^{(t)}[i][j]$ in the bit pipeline tree for level $t$;

if $T_u^{(t)}[i][j1]$ is not empty then
```

```
        make it the right child of $T_{u-1}^{(t)}[i][j]$ in the 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 $T_i$, $0 \leq i < u$. 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 set. In this step
      we will 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 $T_u$. *)

      if $T_u^{(\log n-1)}[0][jr]$ is not empty then
        begin
          make $T_u^{(\log n-1)}[0][jr]$ the child of $T_u^{(\log n)}[0][jr]$ in the random variable tree;

          make $T_u^{(\log n)}[0][jr]$ 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

end
```

Note that i and j in $T_u^{(t)}[i][j]$ are parameters and $T_u^{(t)}$ is not a two dimensional array here. We can view algorithm MRV-Tree as one which distributes random variables x_i into different sets. Each set is indexed by (u, t, i, j). We call these sets BD sets because they are obtained on the bit pipeline trees and the derandomization trees. x is in $BD(u, t, i, j)$ if x is a leaf in $T_u^{(t)}[i][j]$. When u and t are fixed $BD(u, t, i, j)$ sets are disjoint. Because we allow $O(\log n)$ time for each step in MRV-Tree, the time complexity for constructing the random variable trees is $O(\log^2 n)$.

Example: See Fig. 6 for an execution of MRV-Tree. Variables are distributed into the BD sets as shown below.

Step 0 :

$$\begin{aligned} x_0, x_1 &\in BD(0,0,0,\epsilon);\\ x_2, x_3 &\in BD(0,0,1,\epsilon);\\ x_4, x_5 &\in BD(0,0,2,\epsilon);\\ x_6, x_7 &\in BD(0,0,3,\epsilon). \end{aligned}$$

Step 1 :

$$\begin{aligned} x_0, x_1, x_2, x_3 &\in BD(0,1,0,\epsilon);\\ x_4, x_5, x_6, x_7 &\in BD(0,1,1,\epsilon);\\ x_0, x_1 &\in BD(1,0,0,0);\\ x_2, x_3 &\in BD(1,0,1,0);\\ x_4 &\in BD(1,0,2,0);\\ x_5 &\in BD(1,0,2,1);\\ x_6, x_7 &\in BD(1,0,3,0). \end{aligned}$$

Step 2 :

$$\begin{aligned} x_0, x_1, x_2, x_3,&\\ x_4, x_5, x_6, x_7 &\in BD(0,2,0,\epsilon);\\ x_0, x_1 &\in BD(1,1,0,0);\\ x_2, x_3 &\in BD(1,1,0,1);\\ x_4 &\in BD(1,1,1,0);\\ x_5, x_6, x_7 &\in BD(1,1,1,1);\\ x_0, x_1 &\in BD(2,0,0,00);\\ x_2, x_3 &\in BD(2,0,1,00);\\ x_4 &\in BD(2,0,2,00);\\ x_5 &\in BD(2,0,2,10);\\ x_6 &\in BD(2,0,3,00);\\ x_7 &\in BD(2,0,3,01). \end{aligned}$$

Step 3 :

$$\begin{aligned} x_0, x_1, x_2, x_3,&\\ x_4, x_5, x_6, x_7 &\in BD(0,3,0,\epsilon);\\ x_0, x_1, x_5, x_6, x_7 &\in BD(1,2,0,0);\\ x_2, x_3, x_4 &\in BD(1,2,0,1);\\ x_0, x_1 &\in BD(2,1,0,00);\\ x_5, x_7 &\in BD(2,1,0,10);\\ x_6 &\in BD(2,1,0,11);\\ x_2, x_3 &\in BD(2,1,1,00);\\ x_4 &\in BD(2,1,1,10). \end{aligned}$$

Step 4 :

$$\begin{aligned} x_2, x_3, x_4 &\in BD(1,3,0,0);\\ x_2, x_3 &\in BD(2,2,0,00);\\ x_4 &\in BD(2,2,0,01). \end{aligned}$$

Step 5 :

$$x_2, x_3 \in BD(2,3,0,00).$$

Now consider GPC functions of the form $Y_i(x_i)$ and $Y_{i,j}(x_i, x_j)$ except the functions in the vertex cluster term. Our algorithm will distribute these functions into sets $BDF(u,t,i',j')$ by the execution of MRV-Tree, where $BDF(u,t,i',j')$ is essentially the BD set except it is for functions. $Y_{i,j}$ is in $BDF(u,t,i',j')$ iff both x_i and x_j are in $BD(u,t,i',j')$, $\max\{k|$ (the k-th bit of $i\ XOR\ j) = 1\} = t$, $|x_i| > u$ and $|x_j| > u$, where XOR is the bitwise exclusive-or operation, with the exception that all functions belong to $BDF(u, \log n, 0, j')$ for some j'. The condition $\max\{k|$ (the k-th bit of $i\ XOR\ j) = 1\} = t$ ensures that x_i and x_j are in different subtree of the tree rooted at $T_u^{(t)}[i'][j']$. The conditions $|x_i| > u$ and $|x_j| > u$ ensure that x_i and x_j are still valid. The algorithm for the GPC functions for P_u is shown below.

```
Procedure **FUNCTIONS**

  **begin**

    Step $t$ ($0 \le t < u$):

      (* Functions in $BDF(0, t, i', \Lambda)$ reach depth 0 of the bit pipeline tree for level $t$. *)
      Wait for the pipeline to be filled;

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

```
    (* Let $S = BDF(u, t, i', j')$. *)
    if $S$ is not empty then
        begin
            for each GPC function $Y_{i,j}(x_i, x_j) \in S$
                compute the BPC function $Y_{i_u,j_u}(x_{i_u}, x_{j_u})$ with conditional bits set to all 0's;

            (* To fix the random bit in $T_u^{(t)}[i'][j']$. *)
            $T_u^{(t)}[i'][j'] := 0$;
            $F_0 := \sum_{Y_{i,j} \in S} Y_{i_u,j_u}(\Psi(x_i, T_u^{(t)}[i'][j']), \Psi(x_j, T_u^{(t)}[i'][j']))$
                $+ \sum_{Y_{i,j} \in S} Y_{i_u,j_u}(\Psi(x_i, T_u^{(t)}[i'][j']) \oplus 1, \Psi(x_j, T_u^{(t)}[i'][j']) \oplus 1) + VC$;
            (* $VC$ is the function value obtained for functions in the vertex cluster term. We shall
            explain how to compute it later. $\oplus$ is the exclusive-or operation. *)

            $T_u^{(t)}[i'][j'] := 1$;
            $F_1 := \sum_{Y_{i,j} \in S} Y_{i_u,j_u}(\Psi(x_i, T_u^{(t)}[i'][j']), \Psi(x_j, T_u^{(t)}[i'][j']))$
                $+ \sum_{Y_{i,j} \in S} Y_{i_u,j_u}(\Psi(x_i, T_u^{(t)}[i'][j']) \oplus 1, \Psi(x_j, T_u^{(t)}[i'][j']) \oplus 1) + VC$;

            if $F_0 \geq F_1$ then $T_u^{(t)}[i'][j'] := 0$
            else $T_u^{(t)}[i'][j'] := 1$;
            (* The random bit is fixed. *)

            (* To decide whether $Y_{i,j}$ should remain in the pipeline. *)
            for each $Y_{i,j} \in S$
                begin
                    if $\Psi(x_i, T_u^{(t)}[i'][j']) \neq \Psi(x_j, T_u^{(t)}[i'][j'])$ then remove $Y_{i,j}$;
                    (* $Y_{i,j}$ is a zero function in the remaining computation of $P_u$ and also a zero function
                    in $P_v$, $v > u$. *)

                    if $(\Psi(x_i, T_u^{(t)}[i'][j']) = \Psi(x_j, T_u^{(t)}[i'][j'])) \wedge (|x_i| \geq u+1) \wedge (|x_j| \geq u+1)$ then
                        (* Let $b = \Psi(x_i, T_u^{(t)}[i'][j'])$. *)
                        put $Y_{i,j}$ into $BDF(u+1, t, i', j'b)$; (* $Y_{i,j}$ to be processed in $Y_{u+1}$. *)

                end
        end

Step $u + \log n$:

    if $S = BDF(u, \log n, 0, j')$ is not empty then
    (* $S$ is the only set left for this step. *)
```

```
begin
    for each GPC function $Y_{i,j}(x_i, x_j)$ $(Y_i(x_i)) \in S$
        compute the BPC function $Y_{i_u,j_u}(x_{i_u}, x_{j_u})$ $(Y_{i_u}(x_{i_u}))$ with conditional bits set to all 0's;

    (* To fix the random bit in $T_u^{(\log n)}[0][j']$. *)
    $T_u^{(\log n)}[0][j'] := 0$;
    $F_0 := \sum_{Y_{i,j} \in S} Y_{i_u,j_u}(\Psi(x_i, T_u^{(\log n)}[0][j']), \Psi(x_j, T_u^{(\log n)}[0][j']))$
        $+ \sum_{Y_i \in S} Y_{i_u}(\Psi(x_i, T_u^{(\log n)}[0][j'])) + VC$;

    $T_u^{(\log n)}[0][j'] := 1$;
    $F_1 := \sum_{Y_{i,j} \in S} Y_{i_u,j_u}(\Psi(x_i, T_u^{(\log n)}[0][j']), \Psi(x_j, T_u^{(\log n)}[0][j']))$
        $+ \sum_{Y_i \in S} Y_{i_u}(\Psi(x_i, T_u^{(\log n)}[0][j'])) + VC$;

    if $F_0 \geq F_1$ then $T_u^{(\log n)}[0][j'] := 0$
    else $T_u^{(\log n)}[0][j'] := 1$;
    (* The random bit is fixed. *)

    (* To decide whether $T_{i,j}$ should remain in the pipeline. *)
    for each $Y_{i,j} \in S$
        begin
            (* Let $b = T_u^{(\log n)}[0][j']$). $\Psi(x_i, T_u^{(\log n)}[0][j'])$ and $\Psi(x_j, T_u^{(\log n)}[0][j'])$ must be equal
            here. *)
            if $(\Psi(x_i, T_u^{(\log n)}[0][j']) = \Psi(x_j, T_u^{(\log n)}[0][j']) = 0) \wedge ((|x_i| \geq u + 1) \vee (|x_j| \geq u + 1))$
            then
                put $Y_{i,j}$ into $BDF(u + 1, \log n, 0, j'b)$;
            else remove $Y_{i,j}$;
        end

    (* To decide whether $Y_i$ should remain in the pipeline. *)
    for each $Y_i \in S$
        begin
            (* Let $b = T_u^{(\log n)}[0][j']$). *)
            if $(\Psi(x_i, T_u^{(\log n)}[0][j']) = 0) \wedge (|x_i| \geq u + 1)$ then
                put $Y_i$ into $BDF(u + 1, \log n, 0, j'b)$;
            else remove $Y_i$;
        end
end

end
```

The functions being evaluated can also be viewed as being pipelined through the derandomization trees.

There are $O(\log n)$ steps in MRV-Tree and FUNCTIONS, each step takes $O(\log n)$ time and $O((m+n)\log n)$ processors.

Now we describe how the functions in the vertex cluster term are evaluated. Each function $Y_{i,j}(x_i, x_j)$ in the vertex cluster term is defined as $Y_{i,j}(x_i, x_j) = -1$ if the first $|x_i|$ bits of x_i are 0's and the first $|x_j|$ bits of x_j are 0's, and otherwise as $Y_{i,j}(x_i, x_j) = 0$. Let $l(i) = |x_i| - u$. Then $Y_{i_u,j_u}(\Lambda,\Lambda)(0^u, 0^u) = -1/2^{l(i)+l(j)}$ and $Y_{i_u,j_u}(0,0)(0^u,0^u) = -1/2^{l(i)+l(j)-2}$ if $|x_i| > u$ and $|x_j| > u$. Procedure MRV-Tree is executed in parallel for each cluster $C(v)$ to build an *evaluation tree* $TC(v)$ for $C(v)$. An evaluation tree is similar to the random variable tree. The difference between the random variable tree and $TC(v)$ is that the leaves of $TC(v)$ consist of variables from $C(v)$. Let $r = T^{(t)}_{u,v}[i'][j']$ be the root of a subtree T' in $TC(v)$ which is to be constructed in the current step. Let L and R be the left and right subtrees of T', respectively. Let r_L and r_R be the roots of L and R, respectively. At the beginning of the current step L and R have already been constructed. Random variables in the interior nodes of L and R have been fixed. Define $M(x,b) = \sum_{\Psi(i,x)=b} \frac{1}{2^{l(i)}}$, where i's are leaves in the subtree rooted at x. At the beginning of the current step $M(r_L, b)$ and $M(r_R, b)$, $b = 0, 1$, have already been computed and associated with r_L and r_R, respectively. During the current step r_L is made the left child of r and r_R is made the right child of r. Now r is tentatively set to 0 and 1 to obtain the value VC for fixing r in procedure FUNCTIONS. We first compute $VC(v,r)$ for each v. $VC(v,r) = 2\sum_{b=0}^{1} M(r_L, b) M(r_R, b \oplus r)$, where $\oplus$ is the exclusive-or operation. The VC value used in procedure FUNCTIONS is $-\sum_{\{v | T^{(t)}_{u,v}[i'][j'] \text{ is not empty}\}} \frac{d(i)}{2} VC(v, T^{(t)}_{u,v}[i'][j'])$. After setting r we obtain an updated value for $M(r,b)$ as $M(r,b) = M(r_L,b) + M(r_R, b \oplus r)$. If T' has only one subtree then $VC(v,r) = 0$ and $M(r,b)$ need to be computed after r is set.

The above paragraph shows that we need only spend $O(T_{VC})$ operations for evaluating VC for all vertex clusters in a BPC problem, where T_{VC} is the total number of tree nodes of all evaluation trees. T_{VC} is $O(m \log n)$ because there are a total of $O(m)$ leaves and some nodes in the evaluation trees have one child.

We briefly describe the data structure for the algorithm. We build the random variable tree and evaluation trees for P_0. Nodes $T^{(t)}_0[i][\Lambda]$ in the random variable tree and nodes $T^{(t)}_{0,v}[i][\Lambda]$ in the evaluation trees and functions in $BDF(0,t,i,\Lambda)$ are sorted by the pair (t,i). This is done only once and takes $O(\log n)$ time with $O(m+n)$ processors[AKS][C]. As the computation proceeds, the random variable tree and each evaluation tree will split into several trees, each BDF set will split into several sets, one for each distinct conditional bit pattern. A BDF set in P_u can split into at most two in P_{u+1}. Since we allow $O(\log n)$ time for each step, we can allocate memory for the new level to be built in the evaluation trees. We use pointers to keep track of the bit pipeline trees and the evaluation trees. The nodes and functions in the same BD and BDF sets (indexed by the same (u,t,i',j')) should be arranged to occupy consecutive memory cells to facilitate the computation of F_0 and F_1 in FUNCTIONS. These operations can be done in $O(\log n)$ time using $O((m+n)/\log n)$ processors.

It is now straightforward to verify that our algorithm for solving a GPC problem takes $O(\log^2 n)$ time,

$O(\log n)$ time for each of the $O(\log n)$ steps. We note that in each step for each BPC problem we have used $O(m+n)$ processors. This can be reduced to $O((m+n)/\log n)$ processors because in each step $O(m+n)$ operations are performed for each BPC problem. They can be done in $O(\log n)$ time using $O((m+n)/\log n)$ processors. Since we have $O(\log n)$ BPC problems, we need only $O(m+n)$ processors to achieve time complexity $O(\log^2 n)$ for solving one GPC problem.

We use $O((m+n)/\log^{0.5} n)$ processors to solve the first $O(\log^{0.5} n)$ GPC problems in the maximal independent set problem. Recall that the execution of a GPC algorithm will reduce the size of the graph by a constant fraction. For the first $O(\log\log n)$ GPC problems the time complexity is $O(\sum_{i=1}^{O(\log\log n)} \log^{2.5} n/c^i) = O(\log^{2.5} n)$, where $c > 1$ is a constant. In the i-th GPC problem we solve $O(c^i \log^{0.5} n)$ BPC problems in a batch, incurring $O(\log^2 n)$ time for one batch and $O(\log^{2.5} n/c^i)$ time for the $O(\log^{0.5} n/c^i)$ batches. The time complexity for the remaining GPC problems is $O(\sum_{i=O(\log\log n)}^{\log^{0.5} n} \log^2 n) = O(\log^{2.5} n)$.

The Speedup Stage

The input graph here is the output graph from the initial stage. The speedup stage consists of the rest of the GPC problems.

We have to reduce the time complexity for solving one GPC problem to under $O(\log^2 n)$ in order to obtain an $o(\log^3 n)$ algorithm for the maximal independent set problem. After the initial stage, we have a small size problem and we have extra processor power to help us speed up the algorithm.

We redesign the random variable tree T for a BPC problem. We use the design as shown in Fig. 3c in section 2. There are $S = \lceil(\log n + 1)/a\rceil$ blocks in T, where a is a parameter.

We note that the design of T incorporates design techniques from both [H2][HI] and [L2][L3]. The advantage of Han and Igarashi's design[H2][HI] is that random bits can be fixed independently if these bits are at the same level of T. The advantage of Luby's design is that there are fewer random bits in T which is desirable in the the speedup stage of our algorithm for the maximal independent set problem.

Lemma 6: If all random variables in the interior nodes of a proper subtree T' of T are fixed, the random variables x_j at the leaves of T' can only assume two different patterns.

Proof: This is because the random variables from the root of T to the parent of the root of T' are common to all x_j's at the leaves of T'. $\Box$

In fact we have implicitly used this lemma in constructing the bit pipeline tree in the design of our GPC algorithm and in procedure RV-Tree.

The q BPC problems in a GPC problem are divided into $b = q/a$ teams (w.l.g. assuming it is an integer). Team i, $0 \le i < b$, has a BPC problems. Let J_w be w-th team. The algorithm for fixing the random variables for J_w can be expressed as follows.

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

Step $t + w$ ($0 \leq t < S$): Fix random variables in block t in random variable forests for J_w.

Each step will be executed in $O(\log n)$ time. Since there are $O(b + S)$ steps, the time complexity is $O(\log^2 n/a)$ for the above algorithm since $q = O(\log n)$.

For a graph with m edges and n vertices, to fix random bits in block 0 for P_0 we need $2^a(m+n)$ processors to enumerate all possible 2^a bit patterns for the a bits in block 0. To fix the bits in block 0 for P_v, $v < a$, we need $2^{a(v+1)}$ patterns to enumerate all possible $a(v+1)$ bits in block 0 for P_u, $u \leq v$. For each of the $2^{a(v+1)}$ patterns, there are 2^v conditional bit patterns. Thus we need $c^{a^2}(m+n)$ processors for team 0 for a suitable constant c. Although the input to each team may have many conditional bit patterns, it contains at most $O(m+n)$ random variable trees (in the input random variable forest). We need keep working for only those conditional bit patterns which are not associated with empty trees. Thus the number of processors needed for each team is the same because when team J_w is working on block i the bits in block i have already been fixed for teams J_u, $u < w$, and because we keep only nonzero functions. The situation here is similar to the situation in the initial stage. Thus the total number of processors we need for solving one GPC problem in time $O(\log^2 n/a)$ is $c^{a^2}(m+n)\log n/a = O(c^{a^2}(m+n)\log n)$. We conclude that one GPC problem can be solved in time $O(\log^2 n/\sqrt{k})$ with $O(c^k(m+n)\log n)$ processors. Therefore the time complexity for the speedup stage is $O(\sum_{k=1}^{\log n}(\log^2 n/\sqrt{k})) = O(\log^{2.5} n)$.

Theorem 6: There is an EREW PRAM algorithm for the maximal independent set problem with time complexity $O(\log^{2.5} n)$ using $O((m+n)/\log^{0.5} n)$ processors. □

We shall call this algorithm MAX.

Further Improvement

To reduce the processor complexity by another factor of $\log n$ on the CREW model we need only work on the first $O(\log\log n)$ GPC problems. These GPC problems belong to the initial stage.

Consider the first GPC problem. At the beginning of the GPC algorithm all GPC functions (in $BDF(0, t, i, \Lambda)$), nodes in the random variable tree (in $T_0^{(t)}[i][\Lambda]$) and nodes in the evaluation trees (in $T_{0,v}^{(t)}[i][\Lambda]$) will be sorted by the parameter (t, i). This takes $O(m\log n/p + \log n)$ time with p processors. A GPC function f will be passed down the bit pipeline tree in the procedure FUNCTIONS. At each depth of the bit pipeline tree f is involved in a constant number of operations. Thus each GPC function will account for $O(\log n)$ operations, giving a total of $O(m\log n)$ operations. This can be done in time $O(m\log n/p + \log^2 n)$ with p processors. The nodes in the random variable tree and the nodes in an evaluation tree, as they pass done the bit pipeline tree, can be decomposed into several random variable trees and evaluation trees, one for each conditional bit pattern. Each leaf in these trees can be involved in $O(\log n)$ operations in a BPC problem and therefore $O(\log^2 n)$ operations in the GPC problem. This gives time $O(m\log^2 n/p + \log^2 n)$. On the CREW PRAM, we can avoid evaluating nodes in a evaluation tree which has only one child. As long as we only evaluate nodes in the evaluation trees which have two children, the number of operations for evaluating the nodes in an evaluation tree is proportional to the number of leaves in the tree. This helps to cut the time for evaluating the evaluation trees to $O(m/p+\log^2 n)$ for a BPC problem and to $O(m\log n/p+\log^2 n)$ for the GPC problem. However, a node v at level l in an evaluation tree could have its parent $p(v)$ at level $l+c$ with $c>1$ because now we require that $p(v)$ have two children. When $p(v)$ is evaluated, we need the value $\Psi(v,p(v))$. In order to obtain this value we keep updated $\Psi(v,w)$ for all leaves v in a random variable tree and the current node w. The $\Psi(v,w)$ value for the n leaves in the random variable forest for a BPC problem will be updated immediately after the random variables at each level are fixed. This takes $O(n\log n/p)$ time for a BPC problem and $O(n\log^2 n/p)$ time for the GPC problem. In summary, the first GPC problem can be solved in time $O(m\log n/p+n\log^2 n/p+\log^2 n)$ with p processors. If $m>n\log n$ the time will become $O(m\log n/p+\log^2 n)$.

One might argue that since the evaluation trees are built bottom up, if a node is not checked one cannot know whether that node has one or two children. The answer is that we cannot avoid checking whether a node r in an evaluation tree of P_u has one or two children. But if we know r has one child, we can avoid checking r's descendants in the bit pipeline tree, *i.e.*, those nodes in P_v, $v>u$, which are descendants of r in the bit pipeline tree.

Our modified algorithm for the GPC problem will first check whether $m>n\log n$. If $m\le n\log n$ we first construct G' induced by vertices in V with degree no greater than $\log n$. We then solve the maximal independent set problem for G' in time $O(m\log n/p+\log^2 n)$, using an algorithm to be described later. Now the remaining graph can be viewed as satisfying $m>n\log n$ and the rest of the computation takes $O(m\log n/p+\log^2 n)$ time as explained above. We therefore achieve time $O(m\log n/p+\log^2 n\log\log n)$ for the first $O(\log\log n)$ GPC problems. The remaining graph can now be solved by MAX.

We now describe an algorithm for finding a maximal independent set for a graph satisfying $\Delta=O(\log n)$. This algorithm is obtained by using a modified version of our $\Delta+1$ vertex coloring algorithm. We first color the graph with $\Delta+1$ colors and then find a maximal independent set by sequencing through these colors.

Lemma 7[HI]: A BPC problem can be solved by first sorting the input BPC functions into the file-major indexing which takes $O(m\log n/p+\log n)$ time, and then building the derandomization tree and derandomizing the random variables which takes $O(m/p+\log n)$ time, where p is the number of processors used. □

The reason that the computation for a BPC problem takes $O(m/p+\log n)$ time except the sorting step is that the derandomization process can be formulated[HI] as a tree contraction process[MR]. Note that in order to establish Lemma 7 the derandomization tree D should take the form that each interior node of D must have at least two children[HI].

Lemma 8: The $\Delta+1$ vertex coloring problem can be solved in time $O(m\log n/p+\log n(\log\log n)^2)$ using

p processors on the CREW PRAM if $\Delta = O(\log n)$.

Proof: Since $\Delta = O(\log n)$, one GPC problem now contains only $O(\log\log n)$ BPC problems. Since the derandomization trees for all the BPC problems in a GPC problem are the same, we need only build one derandomization tree and then make $O(\log\log n)$ copies of the tree. The time complexity for building the derandomization trees in the GPC algorithm is $O(m\log n/p+\log n)$. The time complexity for building the tables is $O(m\log\log n/p+\log n)$ for a BPC problem because $\Delta = O(\log n)$ and $O(m(\log\log n)^2/p+\log n\log\log n)$ for the GPC problem. The time complexity for the rest of the computation in the GPC algorithm is $O(m\log\log n/p+\log n\log\log n)$ using p processors because the BPC problems are solved sequentially. Thus the first $O(\log\log n)$ GPC problems can be solved in time

$$O\left((\sum_{i=1}^{O(\log\log n)} \frac{m\log n}{c^i p}) + \log n(\log\log n)^2 \right) = O(m\log n/p + \log n(\log\log n)^2)$$

using p processors. Now the graph has size $O(m/\log^2 n)$. It can be colored using the algorithm in section 4. Note again that each GPC problem has only $O(\log\log n)$ BPC problems. □

Theorem 7: There is a CREW PRAM algorithm for the maximal independent set problem with time complexity $O(\log^{2.5} n)$ using $O((m+n)/\log^{1.5} n)$ processors. □

The dominating operations in each step of our maximal independent set algorithm are memory allocation and summation. These operations can be done in time $O(\log n/\log\log n)$ on the CRCW PRAM[P][Re][CV]. Therefore we have,

Corollary: There is a CRCW PRAM algorithm for the maximal independent set problem with time complexity $O(\log^{2.5} n/\log\log n)$ using $O((m+n)\log\log n/\log^{1.5} n)$ processors. □

6 Maximal Matching

Let $N(M) = \{(i,k) \in E,\ (k,j) \in E \,|\, \exists\,(i,j) \in M\}$. A maximal matching can be found by repeatedly finding a matching M and removing $M \cup N(M)$ from the graph.

We adapt Luby's work[L3] to show that after an execution of the GPC procedure a constant fraction of the edges will be reduced.

Let k_i be such that $2^{k_i-1} < 4d(i) \le 2^{k_i}$. Let $q = \max\{k_i | i \in V\}$. Let $\vec{x} =< x_{ij} \in \{0,1\}^q, (i,j) \in E\}$. The length $|x_{ij}|$ of x_{ij} is defined to be $\max\{k_i, k_j\}$. Define

$$
\begin{aligned}
Y_{ij}(x_{ij}) &= \begin{cases} 1 & \text{if } x_{ij}(|x_{ij}|-1)\cdots x_{ij}(0) = 0^{|x_{ij}|} \\ 0 & \text{otherwise} \end{cases} \\
Y_{ij,i'j'}(x_{ij}, x_{i'j'}) &= -Y_{ij}(x_{ij})Y_{i'j'}(x_{i'j'}) \\
B(\vec{x}) &= \sum_{i \in V} \frac{d(i)}{2} \left(\sum_{j \in adj(i)} \left(Y_{ij}(x_{ij}) + \sum_{k \in adj(j), k \neq i} Y_{ij,jk}(x_{ij}, x_{jk}) \right) \right. \\
&\quad \left. + \sum_{j,k \in adj(i), j \neq k} Y_{ij,ik}(x_{ij}, x_{ik}) \right)
\end{aligned}
$$

where $x_{ij}(p)$ is the p-th bit of x_{ij}.

Function B sets a lower bound on the number of edges deleted from the graph[L3] should edge (i,j) be tentatively labeled as an edge in the matching set if $x_{ij} = (0 \cup 1)^{q-|x_{ij}|}0^{|x_{ij}|}$. The following lemma can be proven by following Luby's proof for Theorem 1 in [L1].

Lemma 9: $E[B] \geq |E|/c$ for a constant $c > 0$. □

Function B can be written as

$$
B(\vec{x}) = \sum_{(i,j) \in E} \frac{d(i)+d(j)}{2} Y_{ij}(x_{ij}) + \sum_{j \in V} \sum_{i,k \in adj(j), i \neq k} \frac{d(i)}{2} Y_{ij,jk}(x_{ij}, x_{jk})
$$

$$
+ \sum_{i \in V} \frac{d(i)}{2} \sum_{j,k \in adj(i), j \neq k} Y_{ij,ik}(x_{ij}, x_{ik})
$$

By using the same technique in section 5 we can obtain a CREW algorithm for the maximal matching problem with time complexity $O(\log^2 n)$ using $O(M(n))$ processors. The details of this algorithm are omitted here.

There are two cluster terms in function B. We only need explain how to evaluate the cluster term $\sum_{j \in V} \sum_{i,k \in adj(j), i \neq k} \frac{d(i)}{2} Y_{ij,jk}(x_{ij}, x_{jk})$. The rest of the functions can be computed as we have done for the maximal independent set problem in section 5.

Again we build an evaluation tree $TC(v)$ for each cluster $C(v)$ in the cluster term. Let $l(ij) = |x_{ij}| - u$. Let $r = T_{u,v}^{(t)}[i'][j']$ be the root of a subtree T' in $TC(v)$ which is to be constructed in the current step. Let L and R be the left and right subtrees of T', respectively. Let r_L and r_R be the roots of L and R, respectively. At the beginning of the current step L and R have already been constructed. Random variables in the interior nodes of L and R have been fixed. Define $M(x,b) = \sum_{\Psi(ij,x)=b} \frac{1}{2^{l(ij)}}$. Define $N(x,b) = \sum_{\Psi(ij,x)=b} \frac{d(i)}{2} \frac{1}{2^{l(ij)}}$. At the beginning of the current step $M(r_L, b)$, $M(r_R, b)$, $N(r_L, b)$ and $N(r_R, b)$, $b = 0, 1$, have already been computed and associated with r_L and r_R, respectively. During the current step r_L is made the left child of r and r_R is made the right child of r. Now r is tentatively set to 0 and 1 to obtain value VC for fixing r.

We first compute $VC(v,r)$ for each v. $VC(v,r)=\sum_{b=0}^{1}(N(r_L,b)M(r_R,b\oplus r)+M(r_L,b)N(r_R,b\oplus r))$. The VC value is $-\sum_{\{v|T_{u,v}^{(t)}[i'][j'] \text{ is not empty}\}} VC(v,T_{u,v}^{(t)}[i'][j'])$. After setting r we obtain updated value $M(r,b)$ and $N(r,b)$ as $M(r,b)=M(r_L,b)+M(r_R,b\oplus r)$, $N(r,b)=N(r_L,b)+N(r_R,b\oplus r)$.

Since this computation does not require more processors, we have,

Theorem 8: There is an EREW PRAM algorithm for the maximal matching problem with time complexity $O(\log^{2.5} n)$ using $O((m+n)/\log^{0.5} n)$ processors. $\square$

For the maximal matching problem we cannot remove another factor of $\log n$ from the processor complexity as we did for the maximal independent set problem because there are $O(m)$ leaves in the random variable trees of a BPC problem while there are only $O(n)$ leaves in the maximal independent set problem.

Again in the CRCW PRAM algorithm a factor of $\log\log n$ can be taken out from the time complexity and put into the processor complexity.

Acknowledgement

The author wishes to thank the anonymous referees for their careful reviewing and helpful comments on the manuscript. The author also thanks Professor Richard Cole for making several good suggestions about the paper.

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).

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

[CV] R. Cole and U. Vishkin. Approximate and exact parallel scheduling with applications to list, tree and graph problems. Proc. 27th Symp. on Foundations of Computer Sci., IEEE, 478-491(1986).

[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 ACM Symp. on Theory of Computing, 114-118(1978).

[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).

[HCD] T. Hagerup, M. Chrobak, K. Diks. Optimal parallel 5-coloring of planar graphs. SIAM J. Comput. Vol. 18, No. 2, pp. 288-300(April 1989).

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

[H2] Y. Han. A parallel algorithm for the PROFIT/COST problem. Proc. 1991 Int. Conf. on Parallel Processing, Vol. 3, 103-112(Aug. 1991).

[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).

[II] A. Israeli, A. Itai. A fast and simple randomized parallel algorithm for maximal matching. Computer Science Dept., Technion, Haifa, Israel, 1984.

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

[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. Also in J. of Computer and System Sciences 47, 250-286(1993).

[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).

[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).

[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.

[P] I. Parberry. On the time required to sum n semigroup elements on a parallel machine with simultaneous write. LNCS 227, 296-304.

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

[Re] J. H. Reif. An optimal parallel algorithm for integer sorting. Proc. 26th Symp. on Foundations of Computer Sci., IEEE, 291-298(1985).

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

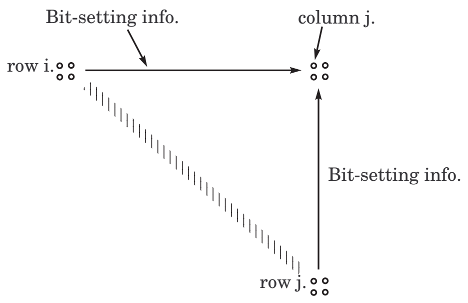


Fig. 1.

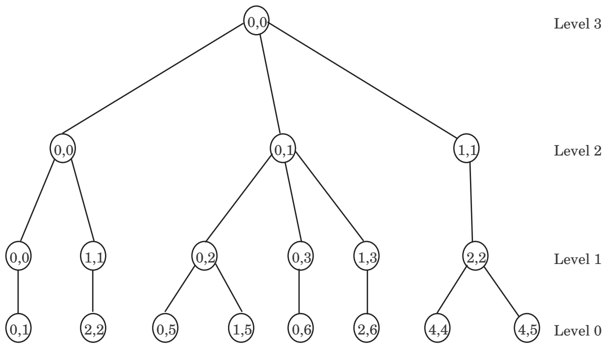


Fig. 2. A derandomization tree. Pairs in the circles are the subscripts of PROFIT/COST problems.

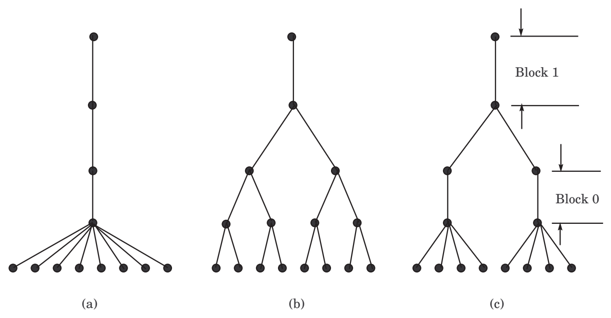


Fig. 3. (a) Luby's tree. (b) Han and Igarashi's tree. (c) A tree of combination.

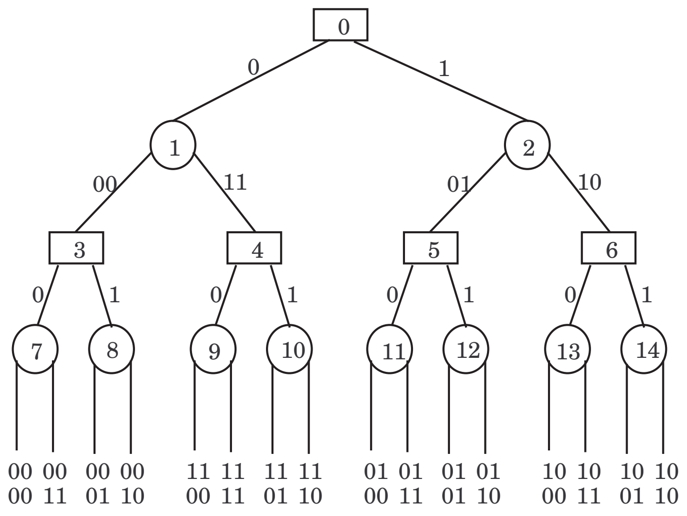
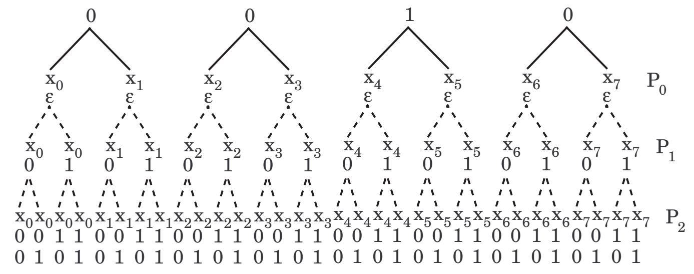
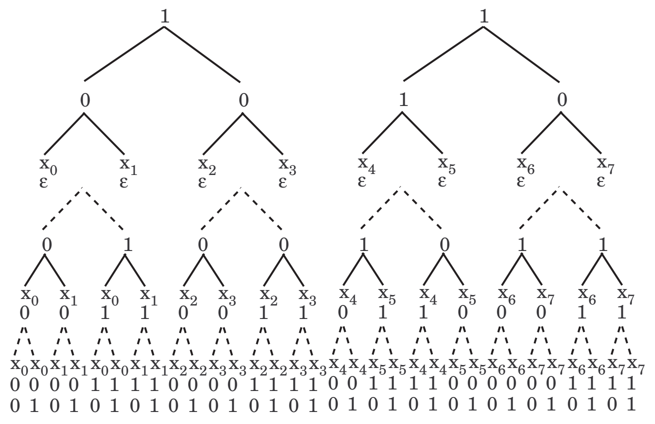


Fig. 4.


(a). Step 0.


(b). Step 1.

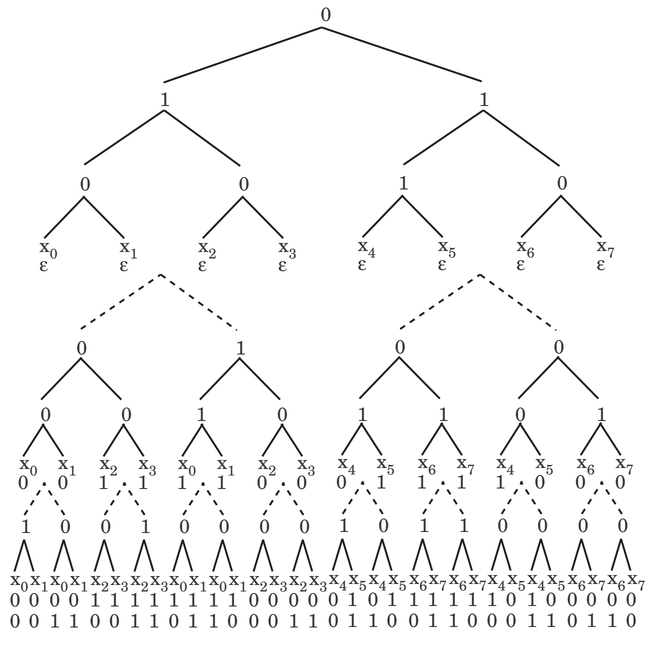
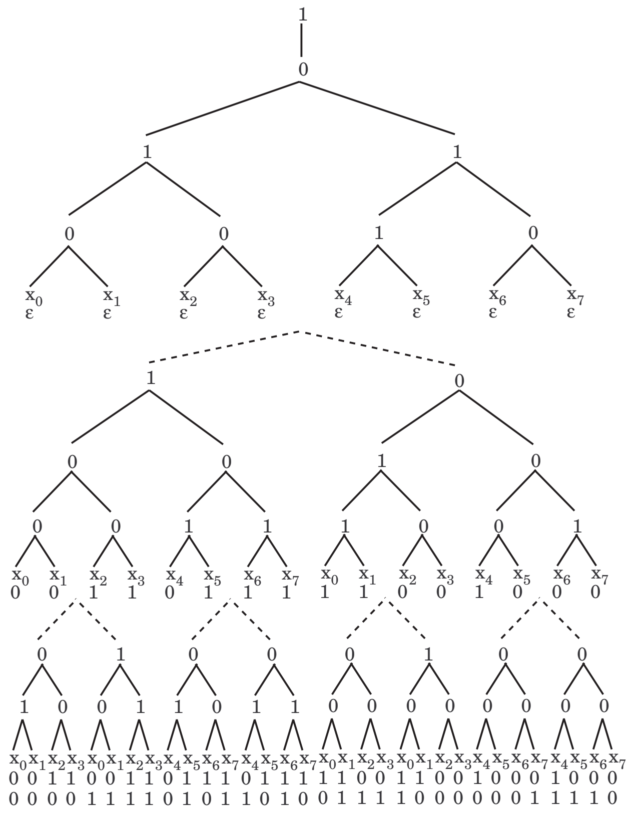
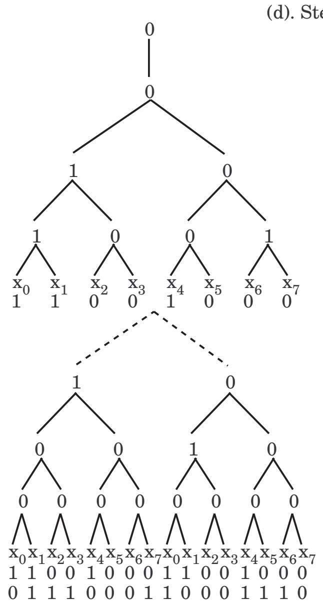

(c). Step 2.

Fig. 5. An execution of RV-Tree. Darkened lines and bits in boldface are random-variable trees. Dotted lines are bit-pipeline trees.


(d). Step 3.


(e). Step 4.

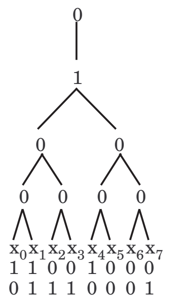
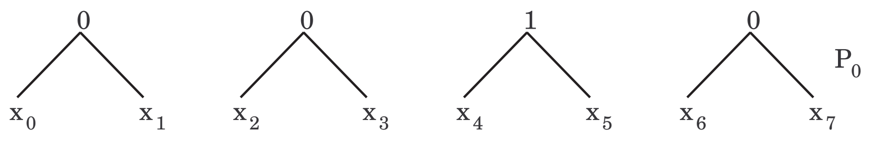
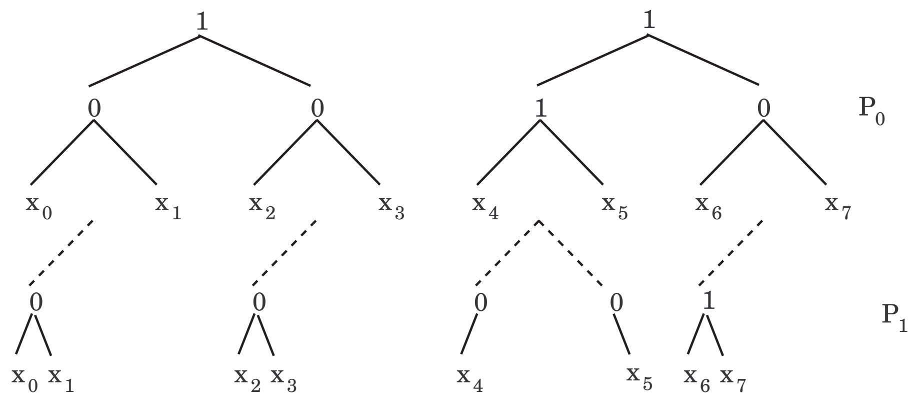

(f). Step 5.

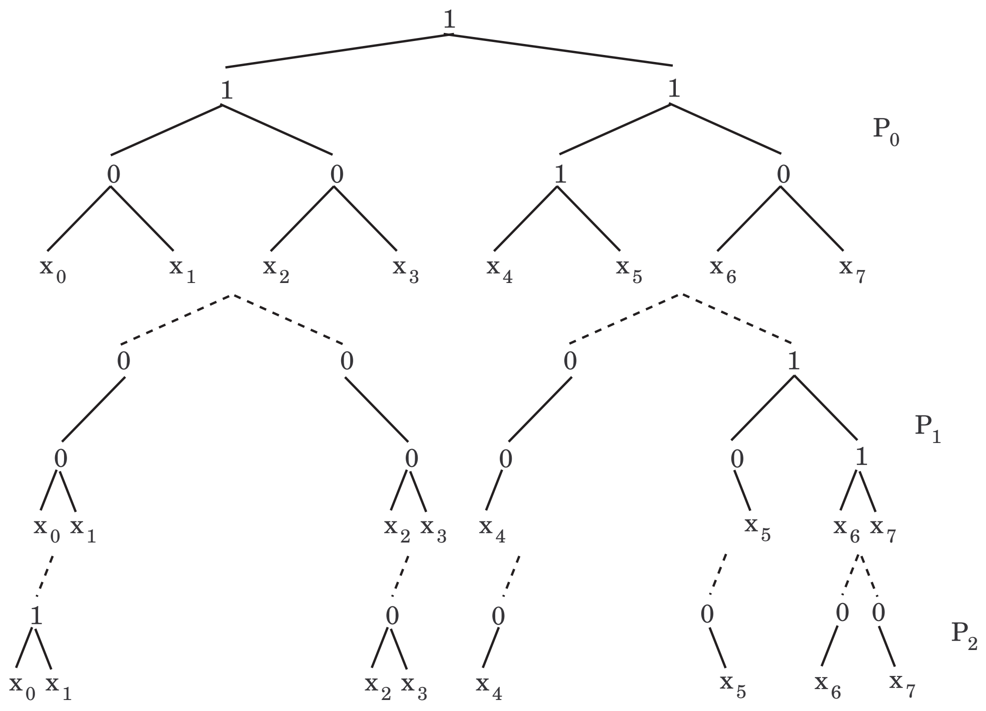
Fig. 5. (cont.)



(a). Step 0.

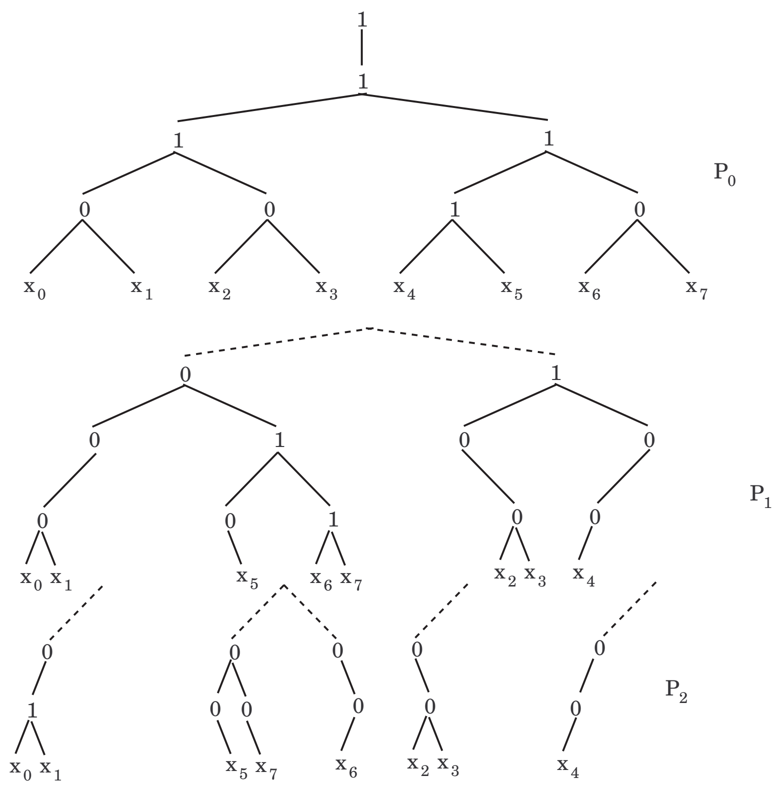


(b). Step 1.

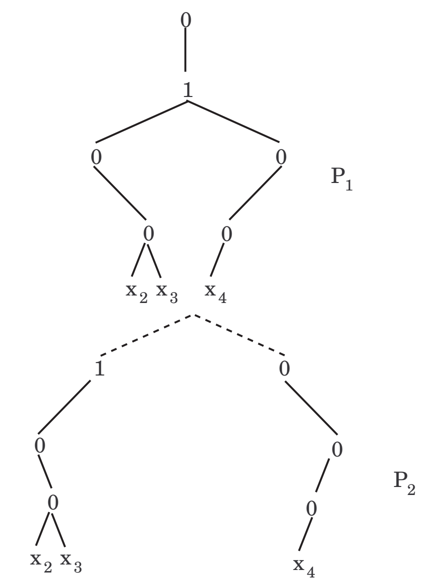


(c). Step 2.

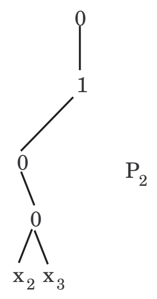
Fig. 6. An execution of MRV-Tree. Darkened lines and bits in boldface are random-variable trees. Dotted lines are bit-pipeline trees.



(d). Step 3.



(e). Step 4.



(f). Step 5.

Fig. 6. (Cont.)