Parallel Algorithms for Maximal Independent Set and Maximal Matching

Yijie Han

School of Computing and Engineering
University of Missouri at Kansas City
Kansas City, MO 64110, USA

Abstract

We present a derandomization process which leads to efficient parallel algorithms for the maximal independent set and maximal matching problems. The derandomization of the general pairs PROFIT/COST problem depends on the derandomization of the bit pairs PROFIT/COST problem which follows Luby's approach of using an $O(n)$ sized sample space. This simplifies the approach presented in [16].

1 Introduction

Maximal independent set and maximal matching are fundamental problems studied in computer science. In the sequential case, a greedy linear time algorithm can be used to solve these problems. However, in parallel computation these problems are not trivial. Currently fastest parallel algorithms for these problems are obtained through derandomization [14][15][16][18]. Due to the complications of the application of derandomization technique, the parallel algorithms obtained [16] were not well understood. We will explain the details of the derandomization technique and its applications to the maximal independent set and maximal matching problem in this book chapter. Instead of using an $O(2^n)$ sized sample space for the bit pairs PROFIT/COST problem as did in [16], we follow Luby's algorithm and use an $O(n)$ sized sample space. This simplifies the approach presented in [16].

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

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

Although the derandomization technique has been applied to the design of sequential algorithms[30] [31], these applications are sequential in nature and cannot be used directly to derandomize parallel algorithms. To apply derandomization techniques to the design of parallel algorithms we have to study how to preserve or exploit parallelism in the process of derandomization. Thus we put emphasis on derandomization techniques which allow us to obtain fast and efficient parallel algorithms.

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

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

The parallel derandomization technique which results in efficient parallel algorithms for maximal independent set and maximal matching originate from Luby's results [25]. Luby formulated maximal independent set and maximal matching problems as PROFIT/COST problems which we will study in detail in the following sections.

we have succeeded in applying PROFIT/COST algorithms in the derandomization of Luby's randomized algorithms for the $\Delta + 1$ vertex coloring problem, the maximal independent set problem and the maximal matching problem and obtained more efficient deterministic algorithms for the three problems[16]. These results are summarized in Table 1.

2 The Bit Pairs PROFIT/COST Probelm

The *Bit Pairs PROFIT/COST* problem (BPC for short) as formulated by Luby[25] can be described as follows.

Let $\vec{x}=< x_i \in \{0,1\} : i = 0, ..., n-1 >$. Each point $\vec{x}$ out of the 2^n points is assigned probability $1/2^n$. Given function $B(\vec{x}) = \sum_{i,j} f_{i,j}(x_i, x_j)$, where $f_{i,j}$ is defined as a function $\{0,1\}^2 \rightarrow \mathcal{R}$. The 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 BENEFIT function and $f_{i,j}$'s are called the PROFIT/COST functions.

The size m of the problem is the number of PROFIT/COST functions present in the input. The input is dense if $m = \theta(n^2)$ and is sparse if $m = o(n^2)$.

The vertex partition problem is a basic problem which can be modeled by the BPC problem[25]. The vertex partition problem is to partition the vertices of a graph into two sets such that the number of edges incident with vertices in both sets is at least half of the number of edges in the graph. Let $G = (V, E)$

Problem	Time Complexity	Processor Complexity	Model	Reference
$\Delta + 1$ vertex coloring	$O(\log^3 n \log\log n)$	$O(m+n)$	CREW	[25][26]
	$O(\log^3 n)$	$O(m+n)$	CREW	[18]
	$O(\log^2 n \log\log n)$	$O((m+n)/\log\log n)$	CREW	[16]
	$O(\log^2 n)$	$O(mn^{\epsilon})$	CREW	[16]
Maximal independent set	$O(\log^3 n)$	$O((m+n)/\log n)$	EREW	[11]
	$O(\log^2 n)$	$O(mn^2)$	EREW	[24]
	$O(\log^2 n)$	$O(n^{2.376})$	CREW	[16]
	$O(\log^{2.5} n)$	$O((m+n)/\log^{0.5} n)$	EREW	[16]
	$O(\log^{2.5} n)$	$O((m+n)/\log^{1.5} n)$	CREW	[16]
Maximal matching	$O(\log^3 n)$	$O(m+n)$	CREW	[19]
	$O(\log^{2.5} n)$	$O((m+n)/\log^{0.5} n)$	EREW	[16]
	$O(\log^2 n)$	$O(n^{2.376})$	CREW	[16]
	$O(\log^4 n)$	$O((m+n)/\log^4 n)$	EREW	[22]
	$O(\log^3 n)$	$O((m+n)/\log^3 n)$	EREW	[17]
	$O(\log^{2.5} n)$	$O((m+n)/\log^{2.5} n)$	EREW	[23]

Table 1.

be the input graph. $|V|$ 0/1-valued uniformly distributed mutually independent random variables are used, one for each vertex. The problem of partitioning vertices into two sets is now represented by the 0/1 labeling of the vertices. Let x_i be the random variable associated with vertex i. For each edge $(i,j) \in E$ a function $f(x_i, x_j) = x_i \oplus x_j$ is defined, where $\oplus$ is the exclusive-or operation. $f(x_i, x_j)$ is 1 iff edge (i,j) is incident with vertices in both sets. The expectation of f is $E[f(x_i, x_j)] = (f(0,0) + f(0,1) + f(1,0) + f(1,1))/4 = 1/2$. Thus the BENEFIT function $B(x_0, x_1, ..., x_{|V|-1}) = \sum_{(i,j)\in E} f(x_i, x_j)$ has expectation $E[B] = \sum_{(i,j)\in E} E[f(x_i, x_j)] = |E|/2$. If we find a good point p in the sample space such that $B(p) \geq E[B] = |E|/2$, this point p determines the partition of vertices such that the number of edges incident with vertices in both sets is at least $|E|/2$.

The BPC problem is a basic problem in the study of derandomization, *i.e.*, converting a randomized algorithm to a deterministic algorithm. The importance of the BPC problem lies in the fact that it can be used as a key building block for the derandomization of more complicated randomized algorithms[16, 26].

Luby[25] gave a parallel algorithm for the BPC problem with time complexity $O(\log^2 n)$ using a linear number of processors on the EREW PRAM model[9]. He used a sample space with $O(n)$ sample points and designed $O(n)$ uniformly distributed pairwise independent random variables on the sample space. His algorithm was obtained through a derandomization process in which a good sample point is found by a binary search of the sample space.

A set of n 0/1-valued uniformly distributed pairwise independent random variables can be designed on a sample space with $O(n)$ points[25]. Let $k = \log n$ (w.l.g. assuming it is an integer). The sample space is $\Omega = \{0, 1\}^{k+1}$. For each $a = a_0 a_1 ... a_k \in \Omega$, $Pr(a) = 2^{-(k+1)}$. The value of random variables x_i, $0 \leq i < n$, on point a is $x_i(a) = (\sum_{j=0}^{k-1}(i_j \cdot a_j) + a_k) \bmod 2$, where i_j is the j-th bit of i starting with the least significant bit. It is not difficult to verify that x_i's are the desired random variables. Because $B(x_0, x_1, ..., x_{n-1}) = \sum_{i,j} f_{i,j}(x_i, x_j)$, where $f_{i,j}$ depends on two random variables, pairwise independent random variables can be used in place of the mutual independent random variables. A good point can be found by searching the sample space. Luby's scheme[25] uses binary search which fixes one bit of a at a time (therefore partitioning the sample space into two subspaces) and evaluates the conditional expectations on the subspaces. His algorithm[26] is shown below.

Algorithm Convert1:
for $l := 0$ **to** k
    begin
        $B_0 := E[B(x_0, x_1, ..., x_{n-1}) \mid a_0 = r_0, ..., a_{l-1} = r_{l-1}, a_l = 0]$;
        $B_1 := E[B(x_0, x_1, ..., x_{n-1}) \mid a_0 = r_0, ..., a_{l-1} = r_{l-1}, a_l = 1]$;
        if $B_0 \geq B_1$ **then** $a_l := 0$ **else** $a_l := 1$;
        /*The value for a_l decided above is denoted by r_l. */
    end
output($a_0, a_1, ..., a_k$);

Since each time the sample space is partitioned into two subspaces the subspace with larger expectation is preserved while the other subspace is discarded, the sample point $(a_0, a_1, ..., a_k)$ found must be a good

point, *i.e.*, the value of B evaluated at $(a_0,\ a_1,\ ...,\ a_k)$ is $\geq E[B]$.

By the linearity of expectation, the conditional expectation evaluated in the above algorithm can be written as $E[B(x_0,\ x_1,\ ...,\ x_{n-1}) \mid a_0 = r_0,\ ...,\ a_l = r_l] = \sum_{i,j} E[f_{i,j}(x_i,\ x_j) \mid a_0 = r_0,\ ...,\ a_l = r_l]$. It is assumed[25] that constant operations(instructions) are required for a single processor to evaluate $E[f_{i,j}(x_i,\ x_j) \mid a_0 = r_0,\ ...\ a_l = r_l]$. Algorithm Convert1 uses a linear number of processors and $O(\log^2 n)$ time on the EREW PRAM model.

Han and Igarashi gave a CREW PRAM algorithm for the PROFIT/COST problem with time complexity $O(\log n)$ using a linear number of processors[18]. They used a sample space of $O(2^n)$ points. The problem is also solved by locating a good point in the sample space. They obtained time complexity $O(\log n)$ by exploiting the redundancy of a shrinking sample space and the mutual independence of random variables.

If we use a random variable tree T_{chain} as shown in Fig. 1 to form the sample space, then we can choose to fix one random bit on the chain in one round as did in Luby's binary search [25], or we could choose to fix more than one random bit in one round. For example we may choose to fix t random bits on the chain in one round. If we choose to do so then in one round we have to enumerate all 2^t possible situations (sub-sample spaces). The total number of rounds we need is $(k+1)/t$. The algorithm where in one round t random bits are fixed is as follows.

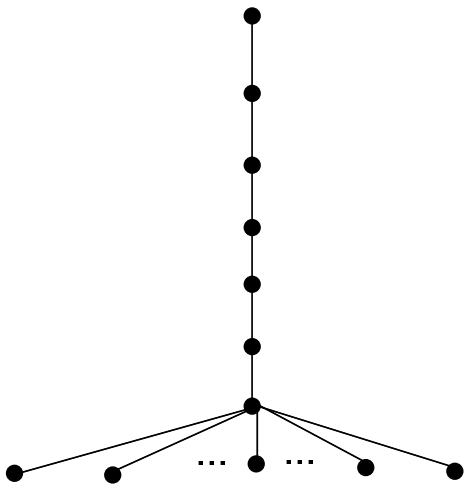


Fig. 1. A random variable tree.

Algorithm Convert2:
for $l := 1$ **to** $(k+1)/t$
begin
$B_i := E[B(x_0,\ x_1,\ ...,\ x_{n-1}) \mid a_0 = r_0,\ ...,\ a_{(l-1)t} = r_{(l-1)t},\ a_{(l-1)t+1}a_{(l-1)t+2}...a_{lt} = i]$, $0 \leq i < 2^t$;
$B_j = \max_{0 \leq i < 2^t} B_i$.
$a_{(l-1)t+1}a_{(l-1)t+2}...a_{lt} = j$;
/*The value for a_i decided above is denoted by r_i, $(l-1)t+1 \leq i \leq lt$. */
end

output(a_0, a_1, ..., a_k);

Algorithm Convert2 involves more operations but has advantage in time. In the case of vertex partitioning problem it can be done in $O((\log^2 n)/t)$ time with $O(2^t m)$ processors.

Let $n = 2^k$ and A be an $n \times n$ array. Elements $A[i, j]$, $A[i, j\#0]$, $A[i\#0, j]$, and $A[i\#0, j\#0]$ form a gang of level 0, which is denoted by $g_A^{(0)}[\lfloor i/2 \rfloor, \lfloor j/2 \rfloor]$. Here $i\#a$ is obtained by complementing the a-th bit of i. All gangs of level 0 in A form array $g_A^{(0)}$. Elements $A[\lfloor i/2^t \rfloor 2^t + a, \lfloor j/2^t \rfloor 2^t + b]$, $0 \leq a, b < 2^t$, form a gang of level t, which is denoted by $g_A^{(t)}[\lfloor i/2^t \rfloor, \lfloor j/2^t \rfloor]$. All gangs of level t in A form array $g_A^{(t)}$.

When visualized on a two-dimensional array A, a step of algorithm Convert1 can be interpreted as follows. Let function $f_{i,j}$ be stored at $A[i, j]$. Setting the random bit at level 0 of the random variable tree is done by examining the PROFIT/COST functions in the diagonal gang of level 0 of A. Setting the random bit at level t of the random variable tree is done by examining the PROFIT/COST functions in the diagonal gang of level t.

A derandomization tree D can be built which reflects the way the BPC functions are derandomized. 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 $A_t[i, j]$ at level $t > 0$ is defined if there exist input functions in the range $A_0[iu, jv]$, $0 \leq u, v < 2^t$. A derandomization tree is shown in Fig. 2. A derandomization tree can be built by sorting the input BPC functions and therefore take $O(\log n)$ time with $m + n$ processors.

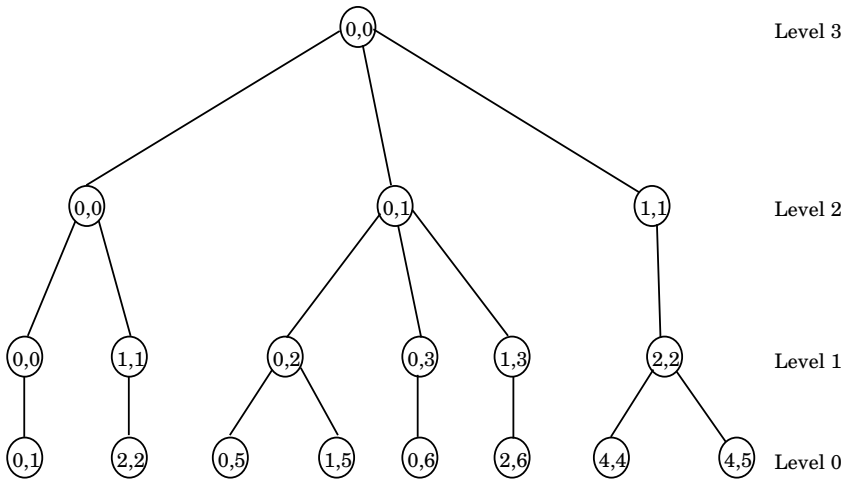


Fig. 2. A derandomization tree. Pairs in circles at the leaves are the subscripts of the BPC functions.

3 The General Pairs PROFIT/COST Problem

Luby [25] formulated the general pairs PROFIT/COST problem (GPC for short) 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 $B(\vec{x}) = \sum_i f_i(x_i) + \sum_{i,j} f_{i,j}(x_i, x_j)$, where f_i is defined as a function $\{0,1\}^q \to \mathcal{R}$ and $f_{i,j}$ is defined as a function $\{0,1\}^q \times \{0,1\}^q \to \mathcal{R}$. The general pairs PROFIT/COST (GPC for short) 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.

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

First we give a sketch of our approach. The incompleteness of the description in this paragraph will be fulfilled later. Let P be the GPC problem we are to solve. P can be decomposed into q BPC problems to be solved sequentially [25]. 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 $f_{i_v,j_v}(x_{i_v}, x_{j_v})$ depends on the setting of bits x_{i_u}, x_{j_u}, $0 \leq u < v$. The main difficulty with our scheme is that when we are working on fixing $\vec{x_v}$, $\vec{x_u}$, $0 \leq u < v$, have not been fixed yet. The only information we can use when we are fixing the random bits at level l of T_u is that random bits 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 in the pipeline. Fortunately this information is sufficient for us to speed up the derandomization process without resorting to too many processors.

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 as the derandomization process proceeds. We use F_u to denote the random variable tree for P_u. We are to fix the random bits on the l-th level of F_v (for $\vec{x_v}$) under the condition that random bits from level 0 to level $l+c-1$, $0 \leq c \leq v$, in F_{v-c} have already been fixed. We are to perform this fixing in $O(\log n)$ time. The random variable trees are built bottom up as the derandomization process proceeds. Immediately before the step we are to fix the random bits on the l-th level of F_u, the random variable trees F_{u-i} have been constructed up to the $(l+i)$-th level. The details of the algorithm for constructing the random variable trees will be given later in this section.

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

Let $a(u:v)$ be the v-th bit to the u-th bit of a. Let $max\_diff(a,b) = t$ mean that t-th bit is the most significant bit that $a(t:t) \neq b(t:t)$.

Lemma 1: In P_d at step $d+t$ there are exactly 2^d conditional bit patterns remain for each BPC function $f_{a,b}(x,y)$ satisfying $max\_diff(a,b) = t$.

Proof: At step $d+t$, the random bits at t-th level in F_v (random variable tree for P_v), $0 \leq v < d$, have been fixed. This limits $x_v y_v$ to two patterns (either 00, 11 or 01, 10). Thus there are 2^d conditional bit patterns $(x_{d-1}, x_{d-2}...x_0, y_{d-1}y_{d-2}...y_0)$ for P_d. □.

The set of remaining conditional bit patterns for $f_{a,b}$ is called the surviving set.

Let a BPC function $f_{a,b}$ in P_d have conditional bit pattern $p_{a,b}(y_{d-1}y_{d-2} \cdots y_0, z_{d-1}z_{d-2} \cdots z_0)$. Let $S_i = \{r | r(\log n : l+1) = i\}$. Let $L_i = \{r | r \in S_i, r(l:l) = 0\}$ and $R_i = \{r | r \in S_i, r(l:l) = 1\}$. We compare

$F_0^{(l)} = \sum_{i \in \{0,1\}^{\log n - l + 1}} \sum_{a \in L_i, b \in R_i} F_{a,b}(\Psi(a(l:0), r^{(l-1:0)}0), \Psi(b(l:0), r^{(l-1:0)}0)) p_{a,b}(y_{d-1}y_{d-2}...y_0, z_{d-1}z_{d-2}...z_0) +$
$F_{b,a}(\Psi(b(l:0), r^{(l-1:0)}0), \Psi(a(l:0), r^{(l-1:0)}0)) p_{b,a}(z_{d-1}z_{d-2}...z_0, y_{d-1}y_{d-2}...y_0)$

with

$F_1^{(l)} = \sum_{i \in \{0,1\}^{\log n - l + 1}} \sum_{a \in L_i, b \in R_i} F_{a,b}(\Psi(a(l:0), r^{(l-1:0)}1), \Psi(b(l:0), r^{(l-1:0)}1)) p_{a,b}(y_{d-1}y_{d-2}...y_0, z_{d-1}z_{d-2}...z_0) +$
$F_{b,a}(\Psi(b(l:0), r^{(l-1:0)}1), \Psi(a(l:0), r^{(l-1:0)}1)) p_{a,b}(y_{d-1}y_{d-2}...y_0, z_{d-1}z_{d-2}...z_0)$

and selects 0 if former is no less than the latter and selects 1 otherwise. Note that $p_{a,b}(y_{d-1}y_{d-2}...y_0, z_{d-1}z_{d-2}...z_0)$ indicates the conditional bit pattern, not a multiplicand. $r^{(l-1:0)}$ is the fixed random bits at levels 0 to $l-1$ in the random variable tree. $\Psi(a,b) = \sum_i a_i \cdot b_i$, where a_i (b_i) is the i-th bit of a (b).

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

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

Step $d+t$ ($0 \leq t < \log n$): 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. At the same time the bit setting information is transmitted to P_{d+1}. *)

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 p_{d+1}.*)

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

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

We now give the algorithm for constructing the random variable trees. This algorithm will help understand better the whole scheme.

The random bit at the l-th level of the random variable trees for P_u is stored in $T_u^{(l)}$. The leaves are stored in $T_u^{(-1)}$. 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 are to build $T_u^{(t)}$. At the beginning of this step $T_{u-1}^{(t)}$ has already been constructed. Let $T_{u-1}^{(t-1)}$ be the child of $T_{u-1}^{(t)}$ in the random variable tree. The setting of the random bit r at level t for P_{u-1}, *i.e.* the random bit in $T_{u-1}^{(t)}$, is known. *)

make $T_u^{(t-1)}$ as the child of
$T_u^{(t)}$ in the random variable tree for P_u;

Fix the random variables in $T_u^{(t)}$;

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 $T_{u-1}^{(\log n)}$ be the root of T_{u-1}. The random bit r in $T_{u-1}^{(\log n)}$ has been fixed.*)

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

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

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

end

4 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[2][10][11][21][24][26] 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[26] formulated each iteration of the while-loop in General-Independent as a GPC problem. We now adapt Luby's formulation[24][26].

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[1]

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

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[24][26] 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 [24] (Theorem 1 in [24]).

Lemma 2[24]: $E[B] \geq |E|/c$ for a constant $c > 0$. □

Function B can be written as

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

where

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

[1] In Luby's formulation[26] $Y_i(x_i)$ 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.

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.

We take advantage of the special properties of the GPC functions to reduce the number of processors to $O(m+n)$. The structure of our algorithm is complicated. We first give an overview of the algorithm.

Overview of the 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 shall call 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 the vertex cluster term. There is a cluster $C(v) = \{x_w | (v,w) \in E\}$ for each vertex v. 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. 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. 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.

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 $C(i)$. When we are fixing j-th random bit the contribution of $C(i)$ can be obtained by evaluating the function $f(x,y)$, where $max\_diff(x,y) = j$. Here $max\_diff(x,y)$ is the most significant bit that x and y differs. If there are a leaves l with $l(j:j) = 0$ and b leaves r with $r(j:j) = 1$ then the contribution from $TC(i)$ for fixing j-th random bit is the sum of ab function values. We will give the details of evaluating this sum using a constant number of operations per cluster.

Let us summarize the main ideas. We 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 view the random variable tree as containing blocks with each block having a levels and a random bits. 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 $O(\log^2 n/a)$ 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 bits for P_v, $0 \le v < u$. We have explained this approach in algorithm Convert2 in section 2. In doing so we would speed up the 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 $O(\log^2 n/a)$. Let b be the number of teams we have. We put all these teams into a pipeline and solve them in time $O((b+\log n/a)\log n)$. 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.

We can view algorithm RV-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 from conditional bit transmission and the derandomization trees. x is in $BD(u, t, i, j)$ if x is a leaf in $T_u^{(t)}$, $x(\log n : t) = i$ and $x(t-1:0) = j$. 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 RV-Tree, the time complexity for constructing the random variable trees is $O(\log^2 n)$.

Example: See Fig. 3 for an execution of RV-Tree. Variables are distributed into the BD sets as shown below.

Step 0 :

$x_0, x_1 \in BD(0,0,0,\epsilon)$;
$x_2, x_3 \in BD(0,0,1,\epsilon)$;
$x_4, x_5 \in BD(0,0,2,\epsilon)$;
$x_6, x_7 \in BD(0,0,3,\epsilon)$.

Step 1 :

$x_0, x_1, x_2, x_3 \in BD(0,1,0,\epsilon)$;
$x_4, x_5, x_6, x_7 \in BD(0,1,1,\epsilon)$;
$x_0, x_1 \in BD(1,0,0,0)$;
$x_2, x_3 \in BD(1,0,1,0)$;
$x_4 \in BD(1,0,2,0)$;
$x_5 \in BD(1,0,2,1)$;
$x_6, x_7 \in BD(1,0,3,0)$.

Step 2 :

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

Step 3 :

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

Step 4 :

$x_2, x_3, x_4 \in BD(1,3,0,0)$;
$x_2, x_3 \in BD(2,2,0,00)$;
$x_4 \in BD(2,2,0,01)$.

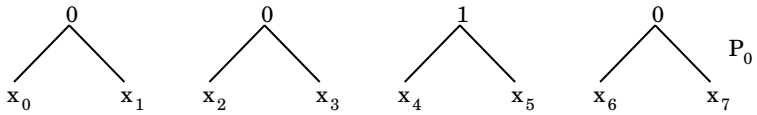
Step 5 :

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

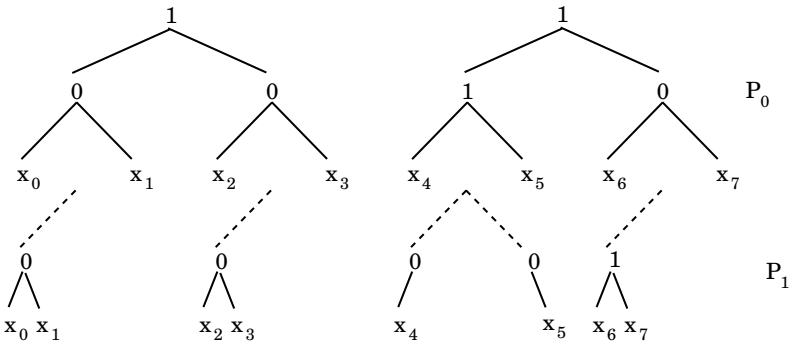
Now consider GPC functions of the form $Y_i(x_i)$ and $Y_{i,j}(x_i, x_j)$ except the functions in the vertex cluster term. Our algorithm will distribute these functions into sets $BDF(u, t, i', j')$ by the execution of RV-Tree, where $BDF(u, t, i', j')$ is essentially the BD set except it is for functions. $Y_{i,j}$ 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 "subtrees". 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**

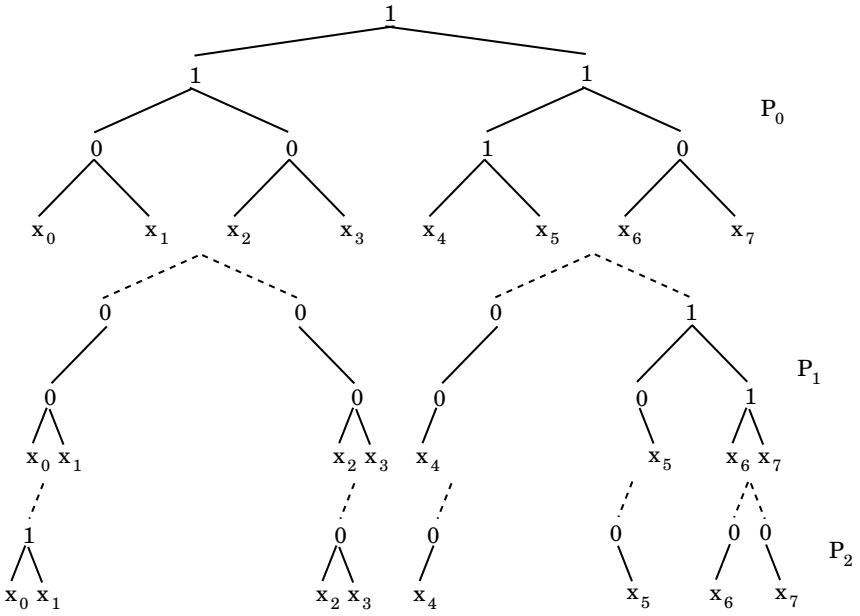
begin



(a). Step 0.

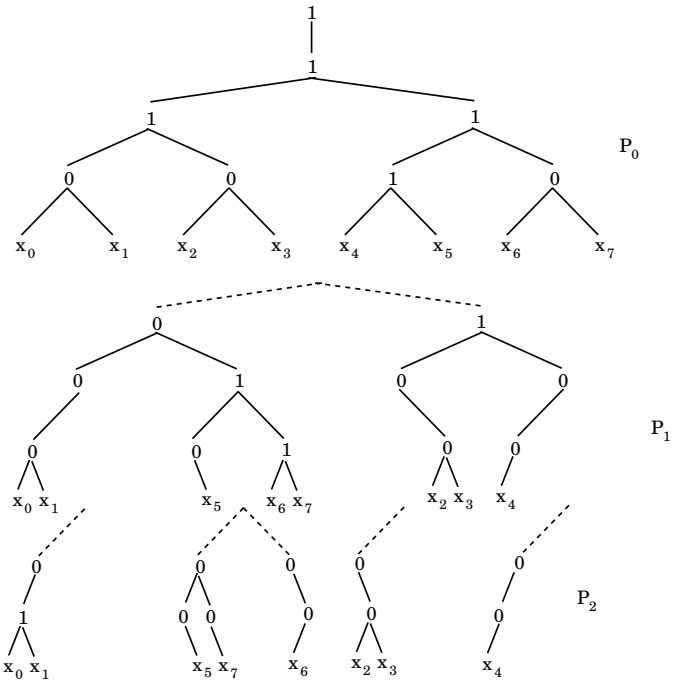


(b). Step 1.

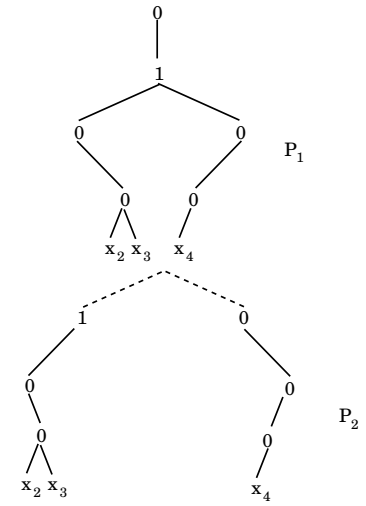


(c). Step 2.

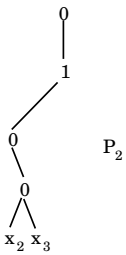
Fig. 3. An execution of RV-Tree. Solid lines depicts the distribution of random variables into BD sets. Dotted lines depicts bit-pipeline trees.



(d). Step 3.



(e). Step 4.



(f). Step 5.

Fig. 3. (Cont.)

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

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

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

(* 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 $T_u^{(t)}$. *)
$T_u^{(t)} := 0$;
$F_0 := \sum_{Y_{i,j} \in S} Y_{i_u,j_u}(\Psi(i(t:0), T_u^{(t:0)}), \Psi(j(t:0), T_u^{(t:0)}))$
$+ \sum_{Y_{i,j} \in S} Y_{i_u,j_u}(\Psi(i(t:0), T_u^{(t:0)}) \oplus 1, \Psi(j(t:0), T_u^{(t:0)}) \oplus 1) + VC$;
(* 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. $T_u^{(t:0)}$ is the random bits in level 0 to t in T_u.*)

$T_u^{(t)} := 1$;
$F_1 := \sum_{Y_{i,j} \in S} Y_{i_u,j_u}(\Psi(i(t:0), T_u^{(t:0)}), \Psi(j(t:0), T_u^{(t:0)}))$
$+ \sum_{Y_{i,j} \in S} Y_{i_u,j_u}(\Psi(i(t:0), T_u^{(t:0)}) \oplus 1, \Psi(j(t:0), T_u^{(t:0)}) \oplus 1) + VC$;

if $F_0 \geq F_1$ **then** $T_u^{(t)} := 0$
else $T_u^{(t)} := 1$;
(* 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 $\Psi(i(t:0), T_u^{(t:0)}) \neq \Psi(j(t:0), T_u^{(t:0)})$ **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 $(\Psi(i(t:0), T_u^{(t:0)}) = \Psi(j(t:0), T_u^{(t:0)})) \wedge (|x_i| \geq u + 1) \wedge (|x_j| \geq u + 1)$ **then**
(* Let $b = T_u^{(t)}$. *)
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 $T_u^{(\log n)}$. *)
$T_u^{(\log n)} := 0$;
$F_0 := \sum_{Y_{i,j} \in S} Y_{i_u,j_u}(\Psi(i(\log n : 0), T_u^{(\log n:0)}), \Psi(x_j(\log n : 0), T_u^{(\log n:0)}))$
$+ \sum_{Y_i \in S} Y_{i_u}(\Psi(i(\log n : 0), T_u^{(\log n:0)})) + VC$;

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

if $F_0 \geq F_1$ **then** $T_u^{(\log n)} := 0$
else $T_u^{(\log n)} := 1$;
(* 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 $b = T_u^{(\log n)}$). $\Psi(x_i(\log n : 0), T_u^{(\log n:0)})$ and $\Psi(x_j(\log n : 0), T_u^{(\log n:0)})$ must be equal
here. *)
if $(\Psi(i(\log n : 0), T_u^{(\log n:0)}) = \Psi(x_j(\log n : 0), T_u^{(\log n:0)}) = 0) \wedge ((|x_i| \geq u+1) \vee (|x_j| \geq u+1))$
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 $b = T_u^{(\log n)}$). *)
if $(\Psi(i(\log n : 0), T_u^{(\log n:0)}) = 0) \wedge (|x_i| \geq u+1)$ **then**
put Y_i into $BDF(u+1, \log n, 0, j'b)$;
else remove Y_i;
end
end

end

There are $O(\log n)$ steps in RV-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 RV-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 $r_j = T^{(j)}_{u,v}$ be the root of a subtree T' in $TC(v)$ which is to be constructed in the current step. Let r_j be at the j-th level. Let $S_i = \{r | r(\log n : j+1) = i\}$. Let $L_i = \{a | a \in S_i,\ a(j:j) = 0\}$ and $R_i = \{a | a \in S_i,\ a(j:j) = 1\}$. At the beginning of the current step L_i and R_i have already been constructed. Random bits at levels less than j have been fixed. Define $M(v, i, v_j, b) = \sum_{\Psi(k(j:0), r(j:0))=b} \frac{1}{2^{l(k)}}$, where k's are leaves in the sub-evaluation-tree rooted at i(i.e. $k(\log n : j+1) = i$). At the beginning of the current step $M(v, i0, r_{j-1}, b)$ and $M(v, i1, r_{j-1}, b)$, $b = 0, 1$, have already been computed. During the current step $i0$'s are at the left side of r_j and $i1$'s are at the right side of r_j. Now r_j is tentatively set to 0 and 1 to obtain the value VC in procedure FUNCTIONS. We first compute $VC(v, r_j)$ for each v. $VC(v, r_j) = 2\sum_{i \in \{0,1\}^{\log n - j}} \sum_{b=0}^{1} M(v, i0, r_{j-1}, b) M(v, i1, r_{j-1}, b \oplus r)$, where $\oplus$ is the exclusive-or operation. The VC value used in procedure FUNCTIONS is $-\sum_{\{v | L_i \text{ or } R_i \text{ is not empty}\}} \frac{d(v)}{2} VC(v, T^{(t)}_{u,v})$. After setting r_j we obtain an updated value for $M(v, i, r_j, b)$ as $M(v, i, r_j, b) = M(v, i0, r_{j-1}, b) + M(v, i1, r_{j-1}, b \oplus r)$. If T' has only one subtree (i.e., either $L_i = \phi$ or $R_i = \phi$) then $VC(v, r) = 0$ and $M(v, i, r_j, b)$ need to be computed after r_j 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 $T^{(t)}_0$ in the random variable tree and nodes $T^{(t)}_{0,v}$ 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[1][5]. As the computation proceeds, 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 conditional bit transmission from P_u to P_{u+1} 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 $S = (\log n + 1)/a$ blocks in T, where a is a parameter.

Lemma 3: If all random bits up to level j are fixed, then random variables in $S_i = \{a | a(\log n : j) = i\}$ can assume only two different patterns.

Proof: This is because the random bits from the root to the $(j+1)$st level are common to all random variables in S_i. □

In fact we have implicitly used this lemma in transmitting conditional bits from P_u to P_{u+1} in the design of our GPC algorithm.

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

Step $t + w$ $(0 \le t < S)$: Fix random variables in block t in random variable trees 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, $u \le v$. 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 $O(m+n)$ BD and BDF sets. We need keep working for only those conditional bit patterns which are not associated with empty BD or BDF sets. 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 2: 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. □

5 Maximal Matching

Let $N(M) = \{(i,k) \in E,\ (k,j) \in E \mid \exists\, (i,j) \in M\}$. 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[26] 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[26] 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 [24].

Lemma 4: $E[B] \ge |E|/c$ for a constant $c > 0$. □

Function B can be written as

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

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

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 $r_j = T^{(t)}_{u,v}$ be the root of a subtree T' in $TC(v)$ at the j-th level which is to be constructed in the current step. Let $S_i = \{a | a(\log n : j+1) = i\}$ Let $L_i = \{a | a \in S_i,\ a(j:j) = 0\}$ and $R_i = \{a | a \in S_i,\ a(j:j) = 1\}$. Let $r_{L_i} = i0$ and $r_{R_i} = i1$ be the roots of L_i and R_i, respectively. At the beginning of the current step L_i's and R_i's have already been constructed. Random bits at level 0 to $j-1$ have been fixed. Define $M(v,i,v_j,b) = \sum_{\Psi(ij(j:0),r(j:0))=b}\frac{1}{2^{l(ij)}}$. Define $N(v,i,v_j,b) = \sum_{\Psi(ij(j:0),r(j:0))=b}\frac{d(v)}{2}\frac{1}{2^{l(ij)}}$. At the beginning of the current step $M(v,i0,v_{j-1},b)$, $M(v,i1,r_{j-1},b)$, $N(v,i0,v_{j-1},b)$ and $N(v,i1,v_{j-1},b)$, $b = 0,1$, have already been computed and associated with $i0$ and $i1$, respectively. During the current step $i0$'s are at the left side of r_j and $i1$'s are at the right side of r_j. Now r_j is tentatively set to 0 and 1 to obtain value VC for fixing r. We first compute $VC(v,r_j)$ for each v. $VC(v,r_j) = \sum_{i\in\{0,`\}^{\log n-j}}\sum_{b=0}^{1}(N(v,i0,v_{j-1},b)M(v,i1,v_{j-1},b\oplus r)+ M(v,i0,v_{j-1},b)N(v,i1,v_{j-1},b\oplus r))$. The VC value is $-\sum_{i|i\in\{0,1\}^{\log n-j},L_i \text{ or } R_i \text{ is not empty}\}} VC(v,r_j)$. After setting r_j we obtain updated value $M(v,i,r_j,b)$ and $N(v,i,r_j,b)$ as $M(v,i,r_j,b) = \sum_i M(v,i0,r_{j-1},b) + M(v,i1,r_{j-1},b\oplus r)$, $N(v,i,r_j,b) = \sum_i N(v,i0,r_{j-1},b) + N(v,i1,v_{j-1},b\oplus r)$.

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

Theorem 3: 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. □

Later development improved the results presented here[17][22][23]. In particular, $O(\log^{2.5n} n)$ time with optimal number of processors ($(m+n)/\log^{2.5} n$ processors) has been achieved [23].

References

[1] 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).

[2] 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).

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

[4] 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).

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

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

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

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

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

[10] 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).

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

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

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

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

[15] Y. Han. A fast derandomization scheme and its applications. Proc. 1991 Workshop on Algorithms and Data Structures, Ottawa, Canada, Lecture Notes in Computer Science 519, 177-188(August 1991).

[16] Y. Han. A fast derandomization scheme and its applications. SIAM J. Comput. Vol. 25, No. 1, pp. 52-82, February 1996.

[17] Y. Han. An improvement on parallel computation of a maximal matching. Information Processing Letters 56 , 343-348(1995).

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

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

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

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

[22] P. Kelson. An optimal parallel algorithm for maximal matching. Information Processing Letters, 52, (1994), 223-228.

[23] KVRCN. Kishore, S. Saxena. An optimal parallel algorithm for general maximal matchings is as easy as for bipartite graphs. Information Processing Letters, 75(4), 145-151(2000).

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

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

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

[27] 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).

[28] 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).

[29] 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.

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

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