Derandomization by Exploiting Redundancy and Mutual Independence

Yijie Han[†] *Yoshihide Igarashi*[‡]

[†]Department of Computer Science
University of Kentucky
Lexington, KY 40506, USA

[‡]Department of Computer Science
Gunma University
Kiryu, 376 Japan

Abstract

We present schemes for derandomizing parallel algorithms by exploiting redundancy of a shrinking sample space and the mutual independence of random variables. Our design uses n mutually independent random variables built on a sample space with exponential number of points. Our scheme yields an $O(\log n)$ time parallel algorithm for the PROFIT/COST problem using no more than linear number of processors.

1 Introduction

Randomization is a powerful tool in the algorithm design. With the aid of randomization the design of algorithms for many difficult problems becomes manageable. This is particularly so in the design of parallel algorithms. Recent progress in this direction not only results in producing many efficient randomized algorithms, but also provides techniques of derandomization, *i.e.*, to convert a randomized algorithm to a deterministic algorithm through a systematic approach of removing randomness. This paper addresses the techniques of derandomization and presents a fast derandomization scheme which yields a fast parallel algorithm for the COST/PROFIT[KW][L2] problem.

A technique of derandomization due to Spencer[Spencer] is to locate a good sample point by an efficient search of the sample space. For an algorithm using n mutually independent random variables the sample space contains exponential number of sample points[ABI]. If every sample point is tested for "goodness" then the algorithm is derandomized with an explosive increase in the time complexity. In order to obtain a polynomial time algorithm, Spencer's scheme uses efficient search techniques to search the sample space for a good point. In the case when the search is binary the sample space is partitioned into two subspaces, and then the subspace with larger probability of finding a good point is preserved while the other discarded. Such a searching technique enables an algorithm to search an exponential sized sample space in polynomial time. Efficient sequential algorithms have been obtain by using this technique[Rag][Spencer].

Spencer's technique seems to be sequential in nature. A known technique[KW][L1] [ABI] to derandomize parallel algorithms is to reduce the size of the sample space by using limited independence. When the sample space contains polynomial number of points, exhaustive search can be used to locate a good point to obtain a DNC algorithm. This technique has been used successfully by Karp and Wigderson[KW], Luby[L1], Alon *et al.*[ABI].

In order to obtain processor efficient parallel algorithms through derandomization, Luby[L2] used the idea of binary search[Spencer] on a sample space with pairwise independent random variables. This technique was developed further by Berger and Rompel[BR] and Motiwani *et al.* [MNN], where $\log^c n$-wise independence among random variables was used. Because the sample space contains $2^{\log^{O(1)} n}$ sample points in the design of [BR][MNN], a thoughtfully designed binary search scheme can still guarantee a DNC algorithm.

We show that a derandomization scheme faster than Luby's[L2] can be obtained. One of our observations leading to a faster derandomization scheme is to exploit the redundancy which is a consequence of a shrinking sample space. Initially a minimum sized sample space is chosen for the design of random variables of limited independence. When a binary search technique is used to search the sample space for a good sample point, the sample space is shrunken or reduced. In fact we observe the shrinkage of the sample space when conditional probability is considered. However, when the sample space is shrunken, the original assumption of independence among random variables can no longer hold. That is, dependency among random variables is expected. Such a dependency is a form of redundancy which can be exploited to the advantage of parallel algorithm design. We note that such redundancy has not been exploited before in the previous derandomization schemes[ABI][KW][L1].

Our scheme uses yet another idea of exploiting mutual independence. Previous derandomization schemes for parallel algorithms tried to stay away from mutual independence because a sample space containing n mutually independent random variables has exponential number of sample points[ABI]. We show, on the contrary, that mutual independence can be exploited for the design of fast parallel algorithms. Our design provides a fast algorithm for the PROFIT/COST[KW][L2] problem with time complexity $O(\log n)$ using no more than linear number of processors. It also improves on the time complexity of the $(\Delta + 1)$-vertex coloring algorithm obtained by Luby[L2].

2 Exploiting Redundancy

We consider the scenario of 0/1-valued uniformly distributed pairwise independent random variables in the following setting.

A set of n 0/1-valued uniformly distributed pairwise independent random variables can be designed on a sample space with $O(n)$ points. The following design is given in [ABI][BR][L2]. Let $k = \lceil \log n \rceil$. The sample space is $\Omega = \{0,\ 1\}^{k+1}$. For each $a = a_0 a_1 ... a_k \in \Omega$, $Pr(a) = 2^{-(k+1)}$. The value of random variables x_i, $0 \le i < n$, on point a is $x_i(a) = (\sum_{j=0}^{k-1}(i_j \cdot a_j) + a_k) \, mod \, 2$.

Typical functions to be searched on have the form $F(x_0, x_1, ..., x_{n-1}) = \sum_{i,j} f_{i,j}(x_i, x_j)$, where $f_{i,j}$ is defined as a function $\{0,\ 1\}^2 \rightarrow \mathcal{R}$. Function F models the important PROFIT/COST problem studied by Karp and Wigderson[KW] and Luby[L2]. Luby's parallel algorithm[L2] for the PROFIT/COST problem has been used as a subroutine in the derandomization of several algorithms[L2][BRS][PSZ].

The problem can be stated as follows. Given function $F(x_0,\ x_1,\ ..,\ x_{n-1}) = \sum_{0 \le i < n, 0 \le j < n} f_{i,j}(x_i,\ x_j)$, find a point $(x_0,\ x_1,\ ...,\ x_{n-1})$ such that $F(x_0,\ x_1,\ ...,\ x_{n-1}) \ge E[F(x_0,\ x_1, ...,\ x_{n-1})]$. Such a point is called a good point. Function F is called the BENEFIT function and functions $f_{i,j}$'s are called the COST/PROFIT functions.

The input of the problem is dense if there are $\Omega(n^2)$ COST/PROFIT functions in F. Otherwise the input is sparse. We denote the number of COST/PROFIT functions in the input by m.

A good point can be found by searching the sample space. Luby's scheme[L2] uses binary search which fixes one bit of a at a time and evaluates the conditional expectations. His algorithm[L2] is shown below.

```
Algorithm Convert1:
for $l := 0$ to $k$
    begin
        $F_0 := E[F(x_0,\ x_1,\ ...,\ x_{n-1}) \mid a_0 = r_0,\ ...,\ a_{l-1} = r_{l-1},\ a_l = 0]$;
        $F_1 := E[F(x_0,\ x_1,\ ...,\ x_{n-1}) \mid a_0 = r_0,\ ...,\ a_{l-1} = r_{l-1},\ a_l = 1]$;
        if $F_0 \ge F_1$ then $a_l := 0$ else $a_l := 1$;
        /*The value for $a_l$ decided above is denoted by $r_l$. */
    end
output($a_0,\ a_1,\ ...,\ a_k$);
```

It is guaranteed that the sample point $(a_0,\ a_1,\ ...,\ a_k)$ found is a good point, *i.e.*, the value of F evaluated at $(a_0,\ a_1,\ ...,\ a_k)$ is $\ge E[F(x_0,\ x_1,\ ...,\ x_{n-1})]$.

By linearity of expectation, the conditional expectation evaluated in the above algorithm can be written as $E[F(x_0, x_1, ..., x_{n-1}) \mid a_0 = r_0, ..., a_l = r_l] = \sum_{i,j} E[f_{i,j}(x_i, x_j) \mid a_0 = r_0,\ ...,\ a_l = r_l]$. We assume that the input is dense, *i.e.*, $m = \Omega(n^2)$. We will drop this assumption in the next section. We also assume 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 $O(n^2 \log n)$ operations. The algorithm can be implemented with $n^2/\log n$ processors and $O(\log^2 n)$ time on the EREW PRAM[BH][S] model.

We observe that as the sample space is being partitioned and reduced, the pairwise independence can no longer be maintained among n random variables. Thus dependency among random variables is expected. Such dependency is a form of redundancy which can be exploited.

After bit a_0 is set, random variables x_i and $x_{i\#0}$ become dependent, where $i\#0$ is obtained by complementing the 0-th bit of i. If a_0 is set to 0 then in fact $x_i = x_{i\#0}$. If a_0 is set to 1 then $x_i = 1 - x_{i\#0}$. Therefore we can reduce n random variables to $n/2$ random variables. Since the input is dense, we are able to cut the number of COST/PROFIT functions from m to about $m/4$.

The modified algorithm is shown below.

```
**Algorithm** Convert2:
**for** $l := 0$ **to** $k$
    **begin**
        $F_0 := \sum_{i,j} E[f_{i,j}(x_i,\, x_j) \mid a_0 = r_0,\, ...,\, a_{l-1} = r_{l-1},\, a_l = 0]$;
        $F_1 := \sum_{i,j} E[f_{i,j}(x_i,\, x_j) \mid a_0 = r_0,\, ...,\, a_{l-1} = r_{l-1},\, a_l = 1]$;
        **if** $F_0 \geq F_1$ **then** $a_l := 0$ **else** $a_l := 1$;
        /* The value for $a_l$ decided above is denoted by $r_l$. */
        combine($f_{i,j}(x_i,\, x_j)$, $f_{i\#l,j}(x_{i\#l},\, x_j)$,
            $f_{i,j\#l}(x_i,\, x_{j\#l})$ and $f_{i\#l,j\#l}(x_{i\#l},\, x_{j\#l})$; for all $i$, $j$);
    **end**
output($a_0$, $a_1$, ..., $a_k$);
```

Some remarks on algorithm Convert2 is in order. The difference between the scheme used in Convert2 and previous schemes is that previous derandomization schemes use a static set of random variables while the set of random variables used in Convert2 changes dynamically as the derandomization process proceeds. Thus our scheme is a dynamic derandomization scheme while the previous schemes are static derandomization schemes.

The redundancy resulting from the shrinking sample space is being exploited resulting in a saving of $O(\log n)$ operations. Algorithm Convert2 can be implemented with $n^2/\log^2 n$ processors while still running in $O(\log^2 n)$ time, since its computing time is bounded by $(c \log^2 n)(1 + \frac{1}{2^2} + \cdots + \frac{1}{2^{2k}})$ for a constant c.

A bit more parallelism can be extracted from algorithm Convert2 by using idling processors to speed up the later steps of the algorithm. When there are $n^2/2^i$ COST/PROFIT functions left, we can extend a by i bits. Thus the number of iterations will be cut down to $O(\log \log n)$. With some modifications of algorithm Convert2, we are able to obtain the time complexity $O(\log n \log \log n)$ using $n^2/\log n \log \log n$ processors.

We note that Luby's technique[L2] can be used here to achieve the same performance. However, there is a conceptual difference between our technique and Luby's, *i.e.*, ours is a dynamic derandomization technique while Luby's is a static one. Our technique uses the idea of exploiting redundancy. This idea is not present in Luby's technique[L2].

3 Exploiting Mutual Independence

In this section we show how mutual independence can be exploited to the advantage of parallel algorithm design. Our idea is embedded in our design of the random variables which is particularly suited to the parallel searching of a good sample point.

In the previous derandomization schemes[ABI][KW][L1] the main objective of the design of random variables is to obtain a minimum sized sample space. Because small sample space requires less effort to search. Original ideas of the design of small sample space for random variables of limited independence can be found in [Bern][Jo][La]. Luby's result[L2] shows that considerations should be given that the design of random variables should facilitate parallel search. Our result presented here carries this idea further in that our design of the random variables facilitates the dynamic derandomization process.

For the problem of finding a good sample point for function $F(x_0,\ x_1,\ ..,\ x_{n-1})= \sum_{i,j} f_{i,j}(x_i,\ x_j)$, Luby's technique of derandomization yields a DNC algorithm with time complexity $O(\log^2 n)$ using linear number of processors. When the input is dense or the COST/PROFIT functions are properly indexed, Luby's technique yields time complexity $O(\log n \log\log n)$ with linear number of processors. However, indexing the functions properly requires $O(\log^2 n \log\log n)$ time with his algorithm[L2].

Previous solutions[L2][BR][MNN] to the problem uses limited independence in order to obtain a small sample space. A small sample space is crucial to the technique of binary searching if DNC algorithms are demanded. In this section we present a case where mutual independence can be exploited to achieve faster algorithms. Since the sample space has exponential number of sample points, binary search can not be used in our situation to yield a DNC algorithm. What happens in our scheme is that the mutual independence helps us to fix random variables independently, thus resulting in a faster algorithm.

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

We now randomize the variables $x_i,\ 0 \le i < n$. Let $r_{i_0},\ r_{i_1},\ ...,\ r_{i_k}$ be the random variables on the path from leaf i to the root of T, where $k = \log n$. Random variable x_i is defined to be $x_i = (\sum_{j=0}^{k-1} i_j \cdot r_{i_j} + r_{i_k})\, mod\, 2$. It can be verified that random variables $x_i,\ 0 \le i < n$ are uniformly distributed pairwise independent random variables. Note the difference between our design and previous designs[ABI][L1][L2].

We shall call tree T the random variable tree.

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

Our algorithm 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 (we shall call this level level 0) and going upward on the tree T until level k. Since there are $k+1$ interior levels in T all random variables will be fixed in $k+1$ steps.

Now consider fixing random variables at level 0. Since there are only two random variables x_j, $x_{j\#0}$ which are functions of random variable r_i (node r_i is the parent of the nodes x_j and $x_{j\#0}$) and x_j, $x_{j\#0}$ are not related to other random variables at level 0, and since random variables at level 0 are mutually independent, they can be fixed independently. This apparently saves computing time because random variables can be fixed locally, thus eliminating the need of collecting global status.

Consider in detail the fixing of r_i which is only related to x_j and $x_{j\#0}$. We simply compute $f_0 = f_{j,j\#0}(0,0)+f_{j,j\#0}(1,1)+f_{j\#0,j}(0,0)+f_{j\#0,j}(1,1)$ and $f_1 = f_{j,j\#0}(0,1)+f_{j,j\#0}(1,0)+f_{j\#0,j}(0,1)+f_{j\#0,j}(1,0)$. If $f_0 \geq f_1$ then set r_i to 0 else set r_i to 1. Our scheme will allow all random variables at level 0 be set in parallel in constant time.

Next we apply the idea of exploiting redundancy. This reduces the n random variables x_i, $0 \leq i < n$, to $n/2$ random variables. COST/PROFIT functions $f_{i,j}$ can also be combined, whenever two functions have the same variables they can be combined into one function. It can be checked that the combining can be done in constant time using linear number of processors and $O(n^2)$ space.

We now have a new function which has the same form of F but has only $n/2$ variables. A recursion on our scheme solve the problem in $O(\log n)$ time with linear number of processors.

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

4 Derandomization using Tree Contraction

In this section we outline further improvements on our derandomization algorithm. We show that the derandomization process can be viewed as a special case of tree contraction[MR]. By using the RAKE operation[MR] we are able to cut down the processor and space complexities further.

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

Obviously we want the input to be arranged by the file-major indexing. When the

0	1	4	5
2	3	6	7
8	9	12	13
10	11	14	15

Fig. 1. File-major indexing.

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

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

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

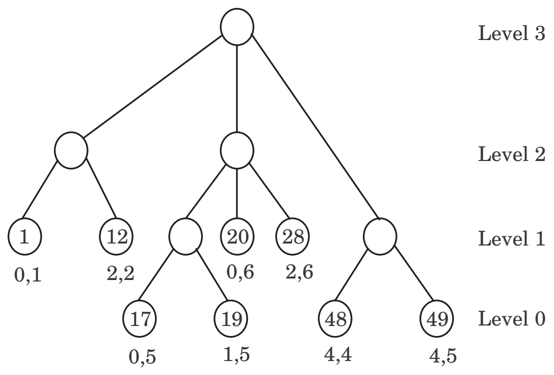
An example of derandomization tree is illustrated in Fig. 2.

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

Since the nodes in the derandomization tree can be sorted by their levels, the derandomization process can be done in $O(\log n)$ time using $m/\log n$ processors.

Theorem 2: A sample point $\vec{r} = (r_0, r_1, ..., r_{n-1})$ satisfying $F|_{\vec{r}} \geq E[F]$ can be found on the CREW PRAM in $O(\log n)$ time using optimal $m/\log n$ processors and $O(m)$ space if the input is arranged by the file-major indexing. $\Box$

We note that if the input is not arranged by the file-major indexing we could sort the input into the file-major indexing. A parallel integer sorting algorithm suffices here.



Pairs under leaves are the subscripts of COST/PROFIT functions. The number in the circles are the corresponding file-major indexing.

Fig. 2. A derandomization tree.

Unfortunately no deterministic parallel algorithm is known for sorting integers of magnitude $n^{O(1)}$ with time complexity $O(\log n)$ using optimal number of processors. We can, of course, use parallel comparison sorting. Known algorithms[AKS][Co] have time complexity $O(\log n)$ using linear number of processors.

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

Our algorithm can be used directly to solve the PROFIT/COST problem in $O(\log n)$ time with optimal number of processors and space if the input is arranged by the file-major indexing. If the file-major indexing for the input is not assumed it would in general use linear number of processors. When our algorithm is used in Luby's $(\Delta + 1)$-vertex coloring algorithm[L2] it will improve the time complexity to $O(\log^3 n)$ from $O(\log^3 n \log\log n)$.

5 Conclusions

Our ideas of exploiting redundancy and mutual independence enables us to obtain a faster derandomization scheme. In principle the idea of exploiting redundancy applies to any search schemes which partitions the sample space. In practice how the redundancy could be exploited depends on each case.

Our scheme could be studied further under more restrictive conditions. Such study would usually yield faster algorithms on the CRCW model, although achievable algorithms would be less general.

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).

[Bern]. S. Bernstein, "Theory of Probability," GTTI, Moscow 1945.

[BH]. R. A. Borodin and J. E. Hopcroft. Routing, merging and sorting on parallel models of computation. Proc. 14th ACM Symp. on Theory of Computing, 1982, pp. 338-344.

[BR]. B. Berger, J. Rompel. Simulating $(\log^c n)$-wise independence in NC. Proc. 1989 IEEE FOCS, 2-7.

[BRS]. B. Berger, J. Rompel, P. Shor. Efficient NC algorithms for set cover with applications to learning and geometry. Proc. 1989 IEEE FOCS, 54-59.

[Co]. R. Cole. Parallel merge sort. 27th Symp. on Foundations of Comput. Sci., IEEE, 511-516(1986).

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

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

[La]. H. O. Lancaster. Pairwise statistical independence, Ann. Math. Stat. 36(1965), 1313-1317.

[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. 1988 IEEE FOCS, 162-173.

[MR]. G. L. Miller and J. H. Reif. Parallel tree contraction and its application. Proc. 26th Symp. on Foundations of Computer Science, IEEE, 291-298(1985).

[MNN]. R. Motwani, J. Naor, M. Naor. The probabilistic method yields deterministic parallel algorithms. Proc. 1989 IEEE FOCS, 8-13.

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

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

[S]. M. Snir. On parallel searching. SIAM J. Comput. 14, 3(Aug. 1985), pp. 688-708.

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