# Determiniistic Unbiased Permuting in Parallel

Gary L. Miller   Shang-Hua Teng

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

March 6, 1989

## 1 Introduction

In this paper, we present a parallel permutation scheme which always halts after $O(\log n)$ steps with a permutation *completely unbaisedly* selected. Our scheme only uses $n$ processors. This improves the previous result of [?] which takes $O(\log n)$ time, using $n^2/\log n$ processors. The models of computation used are Parallel Random Access Machine (PRAM) with finite number of fair coins [?].

The design of many randomized parallel algorithms calls for the efficient generation of uniform random permutation on parallel machines. Basically, there are four types of uniform permutation scheme;

- **Deterministic unbiased scheme**: an algorithm that always halts successfully and each permutation is *compeletely unbiasedly* generated.
- **Random unbiased scheme**: an algorithm that may not halt successfully, but each permutation is *compeletely unbiasedly* generated.
- **Deterministic error bounded scheme**: an algorithm that always halts successfully, and differneces of the probabilities between each pair of permutations are bounded by a small epsilon.
- **Random error bounded scheme**: an algorithm that does not always halt, but differneces of the probabilities between each pair of permutations are bounded by a small epsilon.

Miller and Reif presented an randomized CRCW PRAM parallel algorithm which generates a random permutation of $n$ elements using $O(\log n)$ time, $\frac{n}{\log n}$ processors. However, their algorithm does not always terminate successfully. The probability of failure is $O(\frac{1}{n})$!

Many questions were left open. An interesting and important one is: *is there a parallel deterministic unbiased permutation scheme which runs in $O(\log n)$ time, using polynomial number of processors?.* The affirmative answer to this question provides an efficient and completely trustworthy source for the generating unbiased permutation.

The question was answered affirmatively in [?]. A deterministic $O(\log n)$ time unbiased permutation scheme is derived from their *uniform permutation networks* and *uniform permutation programs.* However, the best know processor count is $n^2/\log n$.

To many applications, such as the parallel load balancing, the time used to generate random permutation is the major concern, provided the time and processor product is linearly bounded by the number of permuted inputs; while the possibility of failure is tolerable if its probability is not terribly big. Therefore the goal is to minimize the time complexity up to the linear time and processor product. We give an optimal randomized PRAM algorithm which generates a random permutation of $n$ elements in $O(\frac{\log n}{\log\log n})$ time, $\frac{n\log\log n}{\log n}$ processors, with the probability of failure at most $O(\frac{1}{n})$. This result improves upon the one of Miller and Reif [?].

## 2 Deterministic Generation Schemes

Let an *$m$-probabilistic-pattern* $\mathcal{P}$ be an $m$-vector $(p_1, \dots, p_m)$ with $1 \leq p_i \leq 1$. An *assignment* $\mathcal{A}$ to an $m$-probabilistic-pattern is an $m$-Boolean-vector $(a_1, \dots, a_m) \in \{0,1\}^M$. The *probability* of an assignment $\mathcal{A} = (a_1, \dots, a_m)$ to a probability-pattern $\mathcal{P} = (p_1, \dots, p_m)$, denoted by $Pr_{\mathcal{P}}(\mathcal{A})$, is

$$Pr_{\mathcal{P}}(\mathcal{A}) = \prod_{i=1}^{M} p_i^{1-a_i}(1-p_i)^{a_i}$$

Let $\mathcal{D} = \{d_1, \dots, d_n\}$ be a finite domain and $\mathcal{Q}_{\mathcal{D}} = (q_1, \dots, q_n)$ be a preassigned probabilistic vector where $0 \leq q \leq 1$ and $\sum_{i=1}^{n} q_i = 1$. $(\mathcal{D}, \mathcal{Q}_{\mathcal{D}})$ is called a *random pair.*

A *deterministic generation scheme* over a finite domain $\mathcal{D}$ is a binary tuple $\mathcal{S} = (\mathcal{P}, \mathcal{F})$ where $\mathcal{P} = (p_1, \dots, p_r)$ is a probabilistic pattern; and $\mathcal{F}$ is a function from $\{0,1\}^r$ to $\mathcal{D}$ where $r$ is callel the *coin complexity* of $\mathcal{S}$.

For all $d \in \mathcal{D}$, let $\mathcal{A}(d)$ be the set of assignments whose image by applying function $\mathcal{F}$ is $d$, i.e.,

$$\mathcal{A}(d) = \mathcal{F}^{-1}(d) = \{\mathcal{A} \mid \mathcal{A} \text{ is a } r\text{-assignment such that } \mathcal{F}(\mathcal{A}) = d\}$$

The *probability* of $d \in \mathcal{D}$ induced by an deterministic permutation scheme $\mathcal{S} = (\mathcal{P}, \mathcal{F})$, denoted by $Prob_{\mathcal{S}}(d)$, is

$$Prob_{\mathcal{S}}(d) = \sum_{\mathcal{A} \in \mathcal{A}(d)} Pr_{\mathcal{P}_n}(\mathcal{A})$$

**Definition 2.1** *$\mathcal{S} = (\mathcal{P}, \mathcal{F})$ is a deterministic generation scheme for a random pair $(\mathcal{D}, \mathcal{Q}_d)$ if for all $i$, $Prob_{\mathcal{S}}(d_i) = q_i$.*

A deterministic generation scheme $\mathcal{S} = (\mathcal{P}, \mathcal{F})$ is *polynomial computable* if $r$ is polynomial in $n$ and $\mathcal{F}$ is a polynomial computable function. A DGS $\mathcal{S} = (\mathcal{P}, \mathcal{F})$ is in $\mathcal{NC}$ if $\mathcal{F}$ is $\mathcal{NC}$ computable.

A *deterministic permutation scheme* is a set $\mathcal{S} = \{\mathcal{S}_1, \dots, \mathcal{S}_\infty\}$, where for all $n$, $\mathcal{S}_n = (\mathcal{P}_n, \mathcal{F}_n)$ is a deterministic generation scheme over domain $S_n$, the set of all permutations of $n$ elements.

**Definition 2.2** *A deterministic permutation scheme $(R, \mathcal{S})$ is* unbiased *if for all $n$, for all $\pi \in S_n$,*

$$Prob_{\mathcal{S}_n}(\pi) = \frac{1}{n!}$$

## 3 Unbiased Permuting and Parallelization

The first deterministic unbiased permutation scheme was published by Moses and Oakford [?] and Durstenfeld [?]. Their algorithm can be specified as following:

```
1. let $A[1 : n]$ be an array with $n$ entries and initially $A[i] = i$;;
2. for $k = n$ to 2
    (a) generate a random number $p_k$ uniformly selected from the set $\{1, \cdots, k\}$;;
    (b) exchange $A[k]$ and $A[p_k]$;;
```

It can be easily prove that the above algorithm is a deterministic unbiased permutation scheme for $n$ elements, and the run time is linear in $n$. The algorithm uses $n-1$ random coins. Moreover, the above algorithm is also a *deterministic generation* scheme in the sense that all random numbers used in the algorithm can be generated at the first step. i.e., the following algorithm.

```
1. generate $n-1$ random numbers $(p_2, \cdots, p_n)$, where $p_k$ is uniformly selected from the
   set $\{1, \cdots, k\}$;;
2. let $A[1 : n]$ be an array with $n$ entries and initially $A[i] = i$;;
3. for $k = n$ to 2
    • exchange $A[k]$ and $A[p_k]$;;
```

At the first glance, the above algorithm is not easy to be parallelized. We shall show that efficient $\mathcal{NC}$ scheme can be derived from a similar transformation.

## 3.1 Inversion Table and Random Permuting

For each permutation $\pi$, let $RANK_\pi[i] = \pi^{-1}(i)$. The *inversion table* $IT_\pi$ of $\pi$ is a $n$-place integral vector defined as,

$$IT_\pi[i] = |\{j \mid j \geq i \ \& \ \pi(j) \leq \pi(i)\}|$$

In other words, $IT_\pi[i]$ is the number of elements in $RANK_\pi$ left to $i$ which are larger than $i$ plus 1. We write $IT(\pi) = IT_\pi$.

**Definition 3.1** *An n-place integral vector T is an* inversion-table *if it is a inversion table of some permutation.*

**Lemma 3.1** *for all n,*

1. *for each n-element permutation $\pi$, $1 \leq IT_\pi[i] \leq n+1-i$;*

2. *if $\pi_1 \neq \pi_2$, then $IT_{\pi_1} \neq IT_{\pi_2}$, i.e., IT is an injective.*

**Lemma 3.2** ([?]) *For each n-element permutation $\pi$, $IT(\pi)$ can be computed in $O(\log n)$ time, using n processors.*

**Lemma 3.3** *An n-place vector T is an* inversion-tables *iff $T[i] \leq n+1-i$.*

[**PROOF**]: the only–if condition follows from the Lemma 3.1, we now prove the sufficient condition.

Let $T$ be an $n$-place integral vector satisfies the condition $T[i] \leq n+1-i$. The following algorithm computes a permutation $\pi$ whose inversion table is $T$.

**Algorithm** $IT^{-1}(T)$

```
1. let $A$ be an $n$-place vector, initially all elements of $A$ are labelled free;;
2. for $k = 1$ to $n$
    (a) $A[j] = k$, where $j$ is the $T[k]^{th}$ free element in $A$;;
3. define a permutation $\pi$ be the one $\pi^{-1}(i) = A[i]$.
```

The correctness of the above algorithm can be easily proved. Clearly, for each $\pi$, $IT^{-1}(IT(\pi)) = \pi$ and for each inversion-tables $T$, $IT(IT^{-1}(T)) = T$. Therefore,

**Lemma 3.4** *IT is a bijection between the set of permutations and the set of inversion-tables.*

The above lemmas suggest the following deterministic unbiased permutation scheme.

**A Perfect Permutation Scheme**

1. *generate* $n-1$ random numbers $(T[1],\cdots,T[n-1])$, where $T[k]$ is uniformly selected from the set $\{1,\cdots,k\}$;;

2. $T[n]=1$;;

3. $\pi = IT{-}1(T)$;;

## 3.2 Conmputing Permutation from Inversion Table in Parallel

For each permutation table $T$, let $\pi = IT^{-1}(T)$. Clearly, $RANK_\pi[1] = T[1]$. Moreover, $RANK_\pi[i]$ is uniquely determined by $(T[1],\cdot,T[i])$.

Let $\oplus$ be a binary operator defined as,

$$a \oplus b = \begin{cases} a & \text{if } a < b \\ a+1 & \text{otherwise} \end{cases}$$

Clearly, $\oplus$ is not associative. Define $\oplus$ to be *left associative*, i.e., $a_1 \oplus a_2 \oplus a_3 = ((a_1 \oplus a_2) \oplus a_3)$. Let

$$\oplus(a_1,\ldots,a_n) = a_1 \oplus a_2 \oplus \cdots \oplus a_n$$

**Lemma 3.5** *For each permutation table $T$, let $\pi = IT^{-1}(T)$,*

1. *$RANK_\pi[1] = T[1]$;*

2. *$RANK_\pi[i] = \oplus(T[i], T[i-1],\cdots,T[1])$.*

Using parallel tree contraction [?,?],

**Lemma 3.6** *for all $a_1,\ldots,a_n$, $\oplus(a_1,\cdots,a_n)$ can be computed in $O(\log n)$ time, using $n$ processors on EREW PRAM.*

**Theorem 3.1** *For each inversion-table $T$, $\pi = IT^{-1}(T)$ can be computed in $O(\log n)$ time, using $n^2$ processors.*

# 4 An Efficient Parallel Permutation Scheme

In this section, we present an $O(\log^2 n)$ time, $O(n)$ processor parallel algorithm for computing a permutation from its inversion table. To the best of our knowledge, this is the first linear processor, NC parallel deterministic unbiased permutation scheme. The efficient parallel algorithm involves the use of some *additive closure properties* of the following unary function class.

## 4.1 A Unary Function Class

For $a_1, \ldots, a_n \in R$, let $f_{(a_1,\ldots,a_n)}$ be a function from $R$ to $R$ such that

$$f_{(a_1,\ldots,a_n)}(x) = \oplus(x, a_n, \cdots, a_1).$$

Let $\mathcal{F}_\oplus$ be the set of all such functions, i.e.,

$$\mathcal{F}_\oplus = \{f_{(a_1,\ldots,a_n)} | n \in \mathcal{N} \text{ and } a_i \in \mathcal{R}\}$$

Clearly, $f_{(a_1,\ldots,a_n)} \circ f_{(b_1,\ldots,b_m)} = f_{(a_1,\ldots,a_n,b_1,\ldots,b_m)}$, in other words, $\mathcal{F}_{oplus}$ is closed under composition. Hence, $(\mathcal{F}, \circ)$ forms a semi-group.

Let $T$ be an inversion-table and $\pi = IT^{-1}(T)$. It follows from Lemma 3.5 that

1. $RANK_\pi[1] = T[1]$;
2. $RANK_\pi[i] = f_{(T[i-1],\cdots,T[1])}(T[i])$.

Let size of a function $f \in \mathcal{F}_{oplus}$, denoted by $size(f)$ be the number of its *breakpoints*, the $x$-value which is not differentiable.

**Lemma 4.1**

1. *All functions from $\mathcal{F}_\oplus$ are montonicly increasing;*
2. *for all $a_1, \ldots, a_n$, $size(f_{(a_1,\ldots,a_n)}) \leq n$;*
3. *for all $f$ and $g \in \mathcal{F}_\oplus$, $size(f \circ g) \leq size(f) + size(g)$.*

It can be proven by induction that each function $f \in \mathcal{F}_\oplus$ of size $n$, there are $b_1 < b_2 \cdots < b_n$ and $c_1 < c_2 \cdots < c_n$ such that,

$$f(x) = \begin{cases} x & x < b_1 \\ x + c_1 & b_1 \leq x < b_2 \\ x + c_2 & b_2 \leq x < b_3 \\ \vdots & \vdots \\ x + c_n & b_n \leq x \end{cases}$$

Such a representation is called a *canonical represention* of $f$.

**Lemma 4.2** *for each function $f, g \in \mathcal{F}_\oplus$ of size $n$ and $m$, respectively, given in canonical form,*

1. *for each $a \in R$, $f(a)$ can be evaluated in $O(\log n)$ time sequentially;*
2. *the canonical representation of $f \circ g$ can be computed in $O(\log(m+n))$ time, using $(m+n)/\log(m+n)$ processors;*

**[PROOF]**: See [?]

## 4.2 An Improved Algorithm

We now show how to use the above lemmas to design a linear processor, NC algorithm to compute a permutation from its inversion table.

The new algorithm consists of two steps, where in the first step, a data structured is built from the given inversion-table which can be used in the second step to compute each $RANK_\pi[i]$ in $O(\log^2 n)$ time sequentially.

The data structure built in the first step is a labeled complete binary tree $B$ of $n-1$ leaves. The nodes of $B$ are named as,

1. The $i^{th}$ leaf is named $v_{i,i}$;
2. an internal node with left child and right child named $v_{i,k}$ and $v_{k+1,j}$, respectively, is named with $v_{i,j}$.

Clearly, the root of $B$ is named with $v_{1,n-1}$.

Let the function of a node $v_{i,j}$, denoted by $F_{v_{i,j}}$, be $f_{(T[i],\cdots,T[j])}$. Clearly, in $O(\log^2 n)$ time, using $n$ processor, the canonical representation of the function of each node in $B$ can be computed.

We now show how to use the information provided in $B$ to compute $IT^{-1}(T)$ in $O(\log^2 n)$ time, using $n$ processors.

Let $p_i = (w_0 w_1 \cdots w_k)$ be the path from leaf $v_{i,i}$ to the root, i.e., $w_0 = v_{i,i}$, $w_k = root$, and $w_{j+1}$ is the parent of $w_j$. In $p_i$, $w_j$ is a *right link* if $w_{i-1}$ is the right child of $w_i$. If $w_j$ is a right link, let $u_j$ be its left child. Then the value of $RANK_\pi[i]$ can be computed by the following procedure.

```
1. let $p_i = (w_0 w_1 \cdots w_k)$ be the path from leaf $v_{i,i}$ to the root;;
2. $value[i] = T[i]$;;
3. for $j = 1$ to $k$
    (a) if $w_j$ is a right link then $value[i] = F_{u_j}(value[i])$;;
4. $RANK_\pi[i] = value[i]$.
```

The correctness of the above procedure can be easily proved. Clearly, using $B$, for each $i$, $RANK_\pi[i]$ can be computed in $O(\log^2 n)$ time, sequentially.

**Theorem 4.1** *A permutation can be computed from its inversion table in $O(\log^2 n)$ time, using $n$ processors. Consequently, there is a deterministic unbiased permutation scheme runs in $O(\log^2 n)$ time, using $n$ processors.*

**Remark 4.1** *Using cascading technique of [?], an $O(\log n)$ factor of runing time can be saved from the above algorithm. Therefore.*

**Theorem 4.2** *A permutation can be computed from its inversion table in $O(\log n)$ time, using $n$ processors. Consequently, there is a deterministic unbiased permutation scheme runs in $O(\log n)$ time, using $n$ processors.*

## 5 Optimal Random Unbiased Permuting

To many applications, such as the parallel load balancing, the time used to generate random permutation is the major concern, provided the time and processor product is linearly bounded by the number of permuted inputs; while the possibility of failure is tolerable if its probability is not terribly big. Therefore the goal is to minimize the time complexity up to the linear time and processor product.

The searching of optimal $O(\log n)$ time parallel algorithm for generating random permutations of $n$ elements was raised as an open question by Vishkin. By reducing the random permuting problem to integer sorting, Reif gave the first optimal $O(\log n)$ time randomized parallel random permuting algorithm. The algorithm is on CRCW PRAM. The probability of failure of Reif's algorithm is bounded by $O(\frac{1}{n^c})$. A very simple $O(\log n)$ time optimal random unbiased scheme is given by Miller and Reif [?]. The probability of failure in their scheme is $O(\frac{1}{n})$.

In this section, we presend an optimal random unbiased parallel permutation scheme which runs in $O(\frac{\log n}{\log\log n})$ time, using $\frac{n \log\log n}{\log n}$ processors, with the probability of failure at most $O(\frac{1}{n})$. This result improves upon the one of Miller and Reif.

### 5.1 The Algorithm of Miller and Reif

Miller and Reif's permutation algorithm is very simple. It uses $2n$ space in the common meomry. Initially, all cells of the $2n$ memory cells are *free*. There are $n/\log n$ processors. Each of them takes care of $\log n$ elements. The elements taken cared of by the $i^{th}$ processor is stored in a queue $Q_i$. The following procedure is performed by each processor.

```
1. for $k = 1$ to $c \log n$
    (a) **if** $Q_i$ is not empty, **select** a number $n_i$ randomly distributed from $[1..2n]$;;
    (b) **if** cell $n_i$ is free, then *move* the head of $Q_i$ to cell $n_i$;;
2. **if** all queues are empty, compute the rank of each element in the common memory via
   prefix sum, and halt successfully;;
3. **else** halt with failure.
```

## 5.2 An Improved Algorithm

The improved permutation algorithm involves the following result due to Cole and Vishkin [?].

**Lemma 5.1 (Cole and Vishkin)** *Prefix sum over integers can be computed in $O(\frac{\log n}{\log\log n})$ time, using $\frac{n\log\log n}{\log n}$ processors on CRCW PRAM.*

At the first glance, it seems that it is very easy to incorporate the above lemma with Miller and Reif's permutation algorithm. But, since the simple space is reduced from $O(\log n)$ to $O(\frac{\log n}{\log\log n})$, the probability of failure will be unbounded. Here, we introduce a two phase algorithm to reduce the probability of failure.