Fast Generation of Random Permutations via Networks Simulation*

Artur Czumaj[1], Przemysława Kanarek[2],
Mirosław Kutyłowski[1], Krzysztof Loryś[2,3]

[1] Heinz Nixdorf Institute and Dept. of Mathematics & Computer Science
University of Paderborn, D-33095 Paderborn, Germany
artur,mirekk@uni-paderborn.de

[2] Institute of Computer Science, University of Wrocław
Przesmyckiego 20, PL-51-151 Wrocław, Poland
pka,lorys@tcs.uni.wroc.pl

[3] Department of Computer Science, University of Trier
D-54286 Trier, Germany

Abstract. We consider the classical problem of generating random permutations with the uniform distribution. That is, we require that for an arbitrary permutation π of n elements, with probability $1/n!$ the machine halts with the ith output cell containing $\pi(i)$, for $1 \leq i \leq n$. We study this problem on two models of parallel computations: the CREW PRAM and the EREW PRAM.

The main result of the paper is an algorithm for generating random permutations that runs in $O(\log\log n)$ time and uses $O(n^{1+o(1)})$ processors on the CREW PRAM. This is the first $o(\log n)$-time CREW PRAM algorithm for this problem.

On the EREW PRAM we present a simple algorithm that generates a random permutation in time $O(\log n)$ using n processors and $O(n)$ space. This algorithm matches the running time and the number of processors used of the best previously known algorithms for the CREW PRAM, and performs better as far as the memory usage is considered.

The common and novel feature of both our algorithms is to design first a suitable random network generating a permutation and then to simulate this network on the PRAM model in a fast way.

1 Introduction

Generating permutations is a fundamental problem studied in theoretical and applied computer science already for few decades. Its first applications assumed generation of all permutations of n elements (see [9] and [17], and the references therein), but since this problem is intractable even for very moderate values of n, methods for generating permutations at random have been also extensively investigated. This task can be formally described as follows:

* Partially supported by KBN grant 8 S503 002 07, TEMPUS project JEP 8145, EU ESPRIT Long Term Research Project 20244 (ALCOM-IT), DFG-Sonderforschungsbereich 376 "Massive Parallelität", and DFG Leibniz Grant Me872/6-1.

Definition 1. A machine M *generates a permutation* π of n elements, if, when halted, the output memory cells $Z_1, \ldots, Z_n$ store, respectively, $\pi(1), \ldots, \pi(n)$.

We say that M *generates (uniformly) random permutations* of n elements, if for every permutation π of n elements M generates π with probability $1/n!$.

Throughout the paper we consider only the problem of generating permutations with the uniform distribution and we skip the word "uniform" while talking about generating random permutations.

Rapidly growing interest in randomized algorithms (see e.g. [13, 14]) adds new applications and new motivations for studying this problem. Now, generating random permutations is a basic element for a large number of randomized sequential and parallel algorithms. For many deterministic algorithms one may find some malicious input data for which the algorithm performs poorly, although it works efficiently on average. Permuting the input randomly may transform a difficult input to a good one (at least on the average) and make it tolerable.

When we consider generation of a "random" permutation, we have to assume that our algorithm uses some random resources. There are two ways of using randomness. An algorithm may use random resources only to define the output permutation so that the distribution of the output permutation is uniform and the running time and space upper bounds hold independently on the random choices made by the algorithm. Such an algorithm is said to be *strong randomized*. If an algorithm ensures the claimed bounds to hold only in the expected case, or to hold with high probability, we call it *weak randomized*.

There are many reasons why permutation generation of strong randomized algorithms are superior of weak randomized ones. When we consider a weak randomized algorithm running with the failure probability $f(n)$, the permutation chosen by such an algorithm is uniform with probability $1 - f(n)$; here $f(n)$ is usually a very small function of the form $f(n) = n^{-c}$ or $f(n) = 2^{-\varepsilon n}$, for constants $c > 1$ and $0 < \varepsilon < 1$. Observe that this might mean that the probability of a single permutation differs from the ideal $\frac{1}{n!}$ by almost $f(n)$. An approach of this kind is acceptable as a component in other randomized algorithms, where such a precision is fully acceptable because it adds only the additive term $f(n)$ to the failure probability. However, there are some critical application (e.g. in cryptography) where such a deviation from the uniform distribution might be dangerous.

Due to results of Gottlieb and Kruskal [7], permutation generation cannot be too fast in models, where a processor may communicate only with a small and fixed number of other processors. Since we are interested in very fast permutation generation, we turn our attention to the PRAM model, for which such communication limitations do not apply. We focus on the PRAMs where each memory cell is of logarithmic size; otherwise, when the size of memory cells is unlimited, one can easily generate random permutation in the running time required for broadcasting.

1.1 Related work

Previous parallel algorithms for generating random permutations have been based on three basic techniques. The first one, called *dart throwing*, consists of two steps. First, the input elements are mapped at random into an array of size $O(n)$. Their order in the array gives an implicit random permutation. Then, the elements are compressed into an array of size n. This technique is especially efficient when used on the CRCW PRAM model, where conflicts during writing and reading are allowed. There has been a sequence of papers using this technique [11, 12, 15] culminating in the papers due to Hagerup [8] and Matias and Vishkin [10], who designed weak randomized algorithms that generate random permutations in $O(\log^* n)$ time on the $O(n/\log^* n)$-processor CRCW PRAM, with high probability.

The second technique is based on *integer sorting*. Each element chooses a key uniformly at random from the set $\{1, \ldots, n\}$. Then the elements are sorted according to the keys' values, which define the relative order of the elements with different keys. Finally, the elements with the same key are randomly permuted using sequential algorithm. This technique was first used by Reif [16], who applied his integer sorting algorithm to generate random permutations in time $O(\log n)$ on the $O(n/\log n)$-processor CRCW PRAM, with high probability. Hagerup [8] borrowed this idea to the EREW PRAM model, and showed that, disregarding a small failure probability, generation of random permutations is reducible to integer sorting. In particular, combining with known integer sorting algorithms, this yields a weak randomized algorithm that runs in $O(\log n)$ time on the n-processor EREW PRAM, or an algorithm running in time $O((\log n)^{5/3}(\log\log n)^{1/3})$ on the EREW PRAM with $O(n/\log n)$ processors. Both these results assume the space of $O(n)$ size is used.

The third technique is to implement in parallel a basic sequential algorithm, which we call Shuffle, due to Durstenfeld [6] (see also [9, page 139] and [17]):

Shuffle:
 `for` i `:=` 1 `to` n `do` $\pi(i)$ `:=` i
 `for` i `:=` 1 `to` n `do`
  κ `:=` random element from $\{i, \ldots n\}$
  exchange the values of $\pi(i)$ and $\pi(\kappa)$

It is well known that this algorithm returns permutations according to the uniform distribution. Anderson [1] used this algorithm in parallel setting and showed that it can be run efficiently on parallel machines with a small number of processors communicating through a common bus. He showed that the main loop of Shuffle can be divided into pieces, each piece executed by a different processor. The ordering in which the resulting permutations are composed affects the permutation generated, but not its probability distribution in a serious way, even if the ordering is determined on-line by an adversary controlling the bus. Hagerup [8] later implemented Shuffle to run in $O(\log n)$ time with n processors and $O(n^2)$ space on the EREW PRAM. He was able to reduce the space used, sacrificing however the running time and/or turning from the EREW to the CREW PRAM model. He presented two algorithms that use $O(n^{1+\varepsilon})$ space for arbitrary fixed $\varepsilon > 0$, one running in $O(\log^2 n)$ time on the $O(n/\log n)$-processor EREW PRAM, and another running in time $O(\log n \log\log\log n)$ on the $O(n/\log\log\log n)$-processor CREW PRAM.

Observe that Shuffle has one very important advantage over the first two techniques: it leads to strong randomized algorithms. Thus up to now, the best strong randomized $O(\log n)$-time algorithm uses n-processors and $O(n^2)$ space on the EREW PRAM. We are not aware of any better strong randomized algorithm existing in the literature even on the CREW PRAM.

1.2 New results

We present two algorithms, one running on the EREW PRAM and the other running on the CREW PRAM. Our first result is an efficient implementation of SHUFFLE.

Theorem 2. *There is a strong randomized EREW PRAM algorithm that generates permutations of n elements uniformly at random in time $O(\log n)$ with n processors and $O(n)$ space.*

This algorithm improves all previously known algorithms for generating random permutations for the CREW and the EREW PRAMs. Comparing to former results, we either reduce space used or make it strong randomized and remove concurrent reads, while the other parameters are not worsened. The algorithm is a simple but efficient implementation of SHUFFLE on the EREW PRAM. It uses the minimum number of $O(\log n!)$ random bits required only to define the output permutation.

Our second and **main result** is the first permutation generation algorithm for the CREW PRAM that runs in sub-logarithmic time.

Theorem 3. *There is a strong randomized CREW PRAM algorithm that generates permutations of n elements uniformly at random in time $O(\log \log n)$ using $O(n^{1+\frac{1}{\log\log n}} \cdot \log n)$ processors.*

Because even with the use of randomization and unbounded number of processors, any CREW PRAM requires $\Omega(\log n)$ time to compute the OR of n bits [4, 5], any non-trivial problem that can be solved on this model in time $o(\log n)$ is of special interest. There are extremely few such algorithms, perhaps the most significant so far was the algorithm for merging two ordered sequences of n elements [2].

While designing our algorithms we introduce a novel technique for generating permutations. We study certain suitably defined layered networks whose main feature is that each level of the network is designed locally and independently of the other levels. The final permutation is defined by the paths from the nodes on the first level to the nodes on the terminal level.

1.3 Basic techniques

Matching networks. It is equivalent to construct a random permutation on n elements or to construct a random perfect matching between two sets of n elements. Simply, if μ is such a perfect matching, then we define $\pi_\mu(i) = j$ if $\mu(a_i) = b_j$.

Our way to obtain a random perfect matching will be through constructing special layered networks, called later *matching networks*. Such a network consists of several levels, each containing n nodes. The directed links of the network form perfect matchings between consecutive levels of the network. Any matching network defines a perfect matching μ between the nodes on the first and the last levels: for a node C on the first level, $\mu(C)$ is the unique node Z on the last level so that there is a path between C and Z. Of course the matchings between the levels must be carefully chosen in order to get finally each permutation with the same probability, and simple enough in order to be easily constructible.

Once we have constructed the network, determining the perfect matching μ can be realized by the *pointer jumping technique*: after step i at every node R in level j (except for the nodes of the last 2^i levels) stores a pointer to the single node S on level $j+2^i$ such that there is a path between R and S. At step $i+1$ the processor attached to R reads the pointer stored in S and copies it to R. The new pointer points to a node S' on level $j+2^{i+1}$ such that there is a path between R and S'.

Obviously, the pointer jumping technique can be performed on EREW PRAM in time logarithmic in the number of levels. One can implement it to run so that the time-processor product equals the number of the nodes in the network.

Shuffle networks. We may consider a special kind of networks for the construction described above. For each consecutive levels L_j, L_{j+1} the perfect matching connects the ith node of L_{j+1} to the ith node of L_j except for at most one pair (k,l) such that a *switch* is applied to the kth and lth nodes: the node k of level L_{j+1} is connected to the node l of L_j and the node l of level L_{j+1} is connected to the node k of L_j.

2 EREW Algorithm

In this section we prove Theorem 2. The algorithm Shuffle can be implemented by constructing a *shuffle network* with n layers, where between layers i and $i+1$ there is a switch (i,k_i), where k_i, $i \le k_i \le n$, is chosen uniformly at random and corresponds to the number κ chosen during the i^{th} iteration of the loop of Shuffle.

To find the permutation defined by such a network we may apply pointer jumping, but it would require $O(n^2)$ processors and space in order to be run in $O(\log n)$ time. We show how to use much less processors and space.

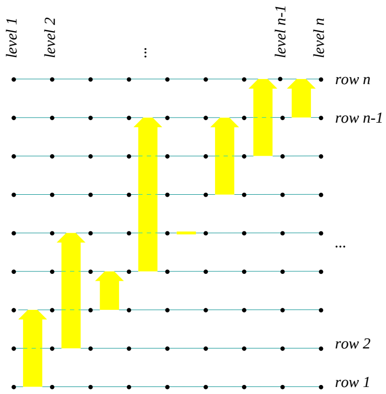

Fig. 1. A shuffle network

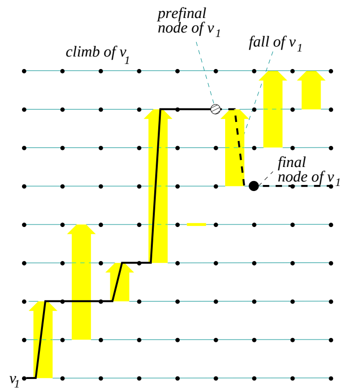

Fig. 2. Paths in a shuffle network

Let the nodes with index i of all levels be called *row* i of the shuffle network. Let v_i denote the path starting at node i at level 1. Each path consists of several parts. First of all it contains edges corresponding to the switches. Such edges may climb or fall: We say that v_i *falls* at level j, if at level j path v_i goes through a switch (j, k_j) and reaches node j at level $j+1$. (It includes the case when $k_j = j$ and the path goes horizontally.) Note that every path falls exactly once. We say that v_i *climbs* at level j, if at level j path v_i goes through a switch (j, k_j) and reaches node k_j at level $j+1$ with $k_j > j$. The remaining edges of the path form a number of horizontal subpaths, each of these subpaths will be called a horizontal segment. More precisely, v_i has a *horizontal segment* between levels $j+1$ and l, if v_i climbs or falls at level l and either $j = 0$ and $i = l$, or v_i climbs at level j with $k_j = l$. Our construction will be based on contracting horizontal segments.

It is convenient to divide each path v_i into two parts. The place where we cut off v_i is the level j where v_i falls. The node of v_i at level j is called *prefinal node* of v_i. Once the prefinal node for each path is known, in one EREW step it is possible to determine the final destination of each path, and thereby the permutation sought.

In order to find the prefinal nodes, we generate a directed graph G consisting of $O(n)$ nodes and $O(n)$ edges. There are n nodes that correspond to the starting positions at level 1, and we denote them by $\langle 0, i\rangle_i$, for $1 \leq i \leq n$. The remaining nodes correspond to the switches (two nodes per switch). There are two nodes corresponding to a switch (i, k_i), $1 \leq i \leq n$, denoted by $\langle i, k_i\rangle_i$ and $\langle i, k_i\rangle_{k_i}$.

There are two kinds of edges in G. First n edges correspond to the edges in the shuffle network where the paths climbs. Thus for each i, $1 \leq i \leq n$, there is a directed edge from $\langle i, k_i\rangle_i$ to $\langle i, k_i\rangle_{k_i}$. The remaining $O(n)$ edges correspond to horizontal segments. We may find these edges by lexicographical sorting triplets (s, i, j) that correspond to the nodes $\langle i, j\rangle_s$. The key observation is that if $\langle j, k_j\rangle_j$ is an intermediate successor of $\langle s, j\rangle_j$ after lexicographical sorting, then there is a horizontal segment between levels $s+1$ and j inside row j. Hence, there is a directed edge from $\langle s, j\rangle_j$ to $\langle j, k_j\rangle_j$ in G. Using the sorting algorithm due to Cole [3] this step can be performed in $O(\log n)$ time on an n processor EREW PRAM with $O(n)$ memory cells.

It follows from the construction that the edges in G determine subpaths of each v_i leading from i to the prefinal node of v_i. What differs the graph G from the shuffle network is that G has only $O(n)$ nodes. So performing pointer jumping on G requires only $O(n)$ processors. Also, since each edge of G corresponds to a subpath of a path in the shuffle network, it has length at most n and pointer jumping takes $O(\log n)$ time. This procedure yields prefinal node for each v_i, and hence by the comments above it also yields the final destination for each v_i. $\square$

Observe that our construction reduces permutation generation to stable integer sorting, or alternatively, to sorting distinct integers drawn from the set $\{1, \ldots, n^2\}$. Apart from integer sorting all other operations can be performed in $O(\log n)$ time on the $O(n/\log n)$-processor EREW PRAM with $O(n)$ space. Thus our construction extends a similar reduction for *weak* randomized permutation generation [8].

3 CREW Algorithm

This section contains a proof of Theorem 3.

Definition 4. We say that a random variable $X \in [0, l]$ has *hypergeometric probability distribution* $\mathrm{Hp}(l, m)$ if

$$\Pr(X = i) = \binom{m}{i}\binom{m}{l-i} / \binom{2m}{l}.$$

The intuition is that we consider a set A of $2m$ elements consisting of two parts A_1, A_2, of m elements each. Then choose uniformly at random l elements of A. Then X is the number of elements taken from A_1.

Our CREW PRAM algorithm requires that each processor can choose randomly an integer from the interval $[0, m]$, $m \leq n$, with hypergeometric distribution $\mathrm{Hp}(l, m)$, $l \leq m$, in a single PRAM step.

3.1 Outline of the algorithm

In the following we assume that n is a power of 2, which significantly simplifies the notation and does not influence the generality of the results obtained. The algorithm that we present yields a random perfect matching between two sets of n elements given by constructing a matching network as described in Section 1.3. The main component of the network is a *splitter* (see Figure 3):

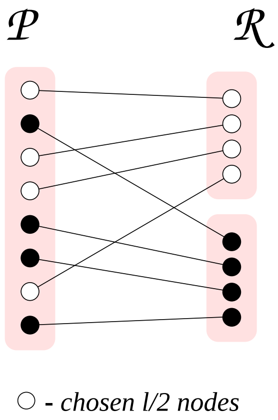

Fig. 3. The matching defined by a splitter

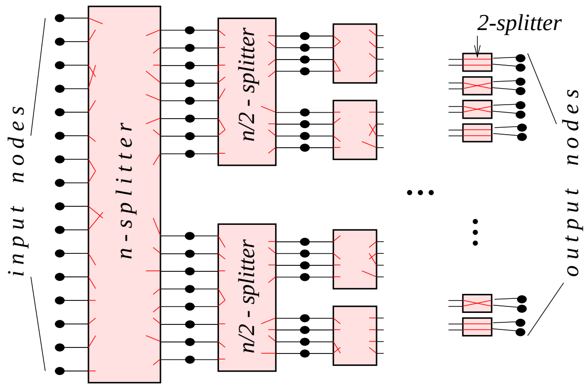

Fig. 4. Structure of the matching network constructed by Algorithm 1

Definition 5. (i) A perfect matching between sequences of nodes $\mathcal{A}$ and $\mathcal{B}$ is called *stable*, if for every $i \leq |\mathcal{A}|$ the ith node of $\mathcal{A}$ is matched with the ith node of $\mathcal{B}$.

(ii) An l*-splitter* between nodes $\mathcal{P} = \{P_0, \ldots, P_{l-1}\}$ and $\mathcal{R} = \{R_0, \ldots, R_{l-1}\}$ is a network which for some sequence $i_0, \ldots, i_{l/2-1}$, $0 \leq i_0 < i_i < \cdots < i_{l/2-1} < l$,

- defines a stable perfect matching between $P_{i_0}, \ldots, P_{i_{l/2-1}}$ and $R_0, \ldots, R_{l/2-1}$,
- defines a stable perfect matching between the sequence of the remaining nodes of $\mathcal{P}$ and $R_{l/2}, \ldots, R_{l-1}$.

The nodes $P_{i_0}, \ldots, P_{i_{l/2-1}} \in \mathcal{P}$ are called later *the chosen nodes of the splitter.*

In the next subsection we show how to construct a random splitter (random means that for a given l and a set $\mathcal{P}'$ of $l/2$ input nodes, the probability of constructing the l-splitter with the chosen nodes $\mathcal{P}'$ equals $\binom{l}{l/2}^{-1}$). Provided that we can build random splitters, the construction of the network defining a random perfect matching may be described as follows:

Algorithm 1 *Recursive construction of random matching network between set of nodes $\mathcal{P} = \{P_0, \ldots, P_{m-1}\}$ and $\mathcal{R} = \{R_0, \ldots, R_{m-1}\}$.*

`if` $m = 1$ `then`
  connect P_0 to R_0
`otherwise`
  let $\mathcal{P}' = \{P'_0, \ldots, P'_{m-1}\}$ be an additional set of nodes.

(1) *Splitting phase:* Choose uniformly at random an m-splitter with the input nodes in $\mathcal{P}$ and the output nodes in $\mathcal{P}'$.
(2) *Recursive call:* Construct random matching networks for input nodes $P'_0, \ldots, P'_{m/2-1}$ and output nodes $R_0, \ldots, R_{m/2-1}$ and, independently, for input nodes $P'_{m/2}, \ldots, P'_{m-1}$ and output nodes $R_{m/2}, \ldots, R_{m-1}$.
(3) Output the composition of networks constructed in (1) and (2).

The above construction may be executed so that the splitting phase and the recursive call are performed in parallel. This rule may be applied to all recursive calls, so finally the PRAM generates, independently and in parallel, a number of splitters. Therefore the time of construction does not exceed the time needed to construct the largest splitter. Moreover, there are $\log n$ stages of recursion, so in order to show that we do not require too many processors it suffices to show that a splitter can be constructed with few processors.

Lemma 6. *Algorithm 1 returns each perfect matching with the same probability.*

Proof. We prove by induction on n that the probability that Algorithm 1 constructs a given perfect matching μ of n elements equals $1/n!$. For $n = 1$ it is obvious. So let $n > 1$ be a power of 2. In order to become μ the following three events must take place. First, during the splitting phase we have to choose $P_{i_0}, \ldots, P_{i_{n/2-1}}$ as exactly these nodes that are to be matched with $R_0, \ldots, R_{n/2-1}$ and take a splitter that connects $P_{i_0}, \ldots, P_{i_{n/2-1}}$ with $P'_0, \ldots, P'_{n/2-1}$. Since each splitter is chosen with the same probability, the probability of this event equals $\binom{n}{n/2}^{-1}$. Second, at Phase 2 we have to match the nodes $P'_0, \ldots, P'_{n/2-1}$ with $R_0, \ldots, R_{n/2-1}$ in a unique way in order to get μ. By the induction hypothesis this happens with probability $1/(n/2)!$. Third, at Phase 2 we have to match the nodes $P'_{n/2}, \ldots, P'_{n-1}$ with $R_{n/2}, \ldots, R_{n-1}$ in a unique way, and it happens with probability $1/(n/2)!$. Hence we get matching μ with probability

$$\frac{1}{\binom{n}{n/2}} \cdot \frac{1}{(n/2)!} \cdot \frac{1}{(n/2)!} = \frac{1}{n!}.$$

3.2 Construction of a random splitter

For each i, $0 \le i \le \log n$, let the sets $I_{i,j}$, $0 \le j < 2^i$, partition $\mathcal{I} = \{0, 1, \ldots, n-1\}$ into 2^i consecutive intervals of length $n/2^i$. That is, we put $I_{i,j} = \{j \cdot \frac{n}{2^i}, \ldots, (j+1) \cdot \frac{n}{2^i} - 1\}$. In particular, $I_{\log n, j} = \{j\}$. Let $\mathcal{P}_{i,j} = \{P_l : l \in I_{i,j}\}$.

The idea of the construction of a splitter is that in order to choose $n/2$ input nodes we determine instead how many elements chosen are inside each set $\mathcal{P}_{i,j}$. For this purpose we use a tree $\mathcal{T}$ with the set of vertices $\{T_{i,j} \mid 0 \le i \le \log n \text{ and } 0 \le j < 2^i\}$. We adopt the convention that $T_{i,j}$ is a parent of $T_{i+1,2j}$ and $T_{i+1,2j+1}$, so $\mathcal{T}$ is a binary tree of depth $\log n$. Each vertex $T_{i,j} \in \mathcal{T}$ corresponds to the set $\mathcal{P}_{i,j}$ of input nodes. Thus the root of the tree $T_{0,0}$ corresponds to the set of all input nodes. Further, for $\tau \in \mathcal{T}$ the children of τ correspond to two "halves"of the set of their parent. Finally, each leaf of $\mathcal{T}$ corresponds to a single input node.

We use a labeling cnt of vertices of $\mathcal{T}$ such that $\mathrm{cnt}(T_{i,j})$ equals the number of elements chosen in $\mathcal{P}_{i,j}$. Therefore function cnt has to satisfy the following conditions:

- $\mathrm{cnt}(T_{0,0}) = \frac{n}{2}$,
- $\mathrm{cnt}(T_{i,j}) = \mathrm{cnt}(T_{i+1,2j}) + \mathrm{cnt}(T_{i+1,2j+1})$,
- $0 \le \mathrm{cnt}(T_{i,j}) \le |I_{i,j}|$,

for $0 \le i < \log n$ and $0 \le j \le 2^i$. If these properties hold, then $\langle \mathcal{T}, \mathrm{cnt} \rangle$ is called a *distribution tree*. We say also that a node P_j is *chosen*, if and only if $\mathrm{cnt}(T_{\log n, j}) = 1$.

How to construct a distribution tree efficiently will be discussed in the next subsections. At this moment we assume we are given some distribution tree $\langle \mathcal{T}, \mathrm{cnt} \rangle$ and point how we construct the corresponding splitter.

The set of chosen elements (and hence the distribution tree) defines a unique *splitting perfect matching* μ. Namely, $\mu(P_i) = R_l$ where

$$l = \begin{cases} |\{j \mid j < i \text{ and } P_j \text{ is chosen}\} & \text{if } P_i \text{ is chosen} \\ \frac{n}{2} + |\{j \mid j < i \text{ and } P_j \text{ is not chosen}\} & \text{if } P_i \text{ is not chosen} \end{cases}$$

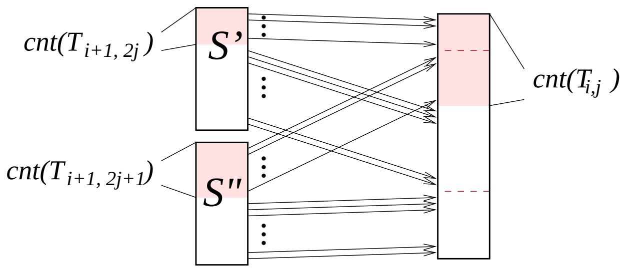

Fig. 5. Matching constructed by the distribution phase

Obviously, μ has all properties demanded from the matching defined by a splitter. Hence it remains only to indicate how we generate a splitter realizing μ. This network can be described recursively as follows:

Algorithm 2 *Recursive description of a splitter*

`if` $n = 1$ `then`
  connect the output node with the input node
`otherwise`
  let $\{P'_0, \ldots, P'_{n-1}\}$ be an additional set of nodes divided into two equal halves S' and S''.

  (1) *Distribution phase:* (see Figure 5).
  - Connect the first $\mathrm{cnt}(T_{1,0})$ output nodes with the first $\mathrm{cnt}(T_{1,0})$ nodes of S',
  - Connect the next $\mathrm{cnt}(T_{1,1})$ output nodes with the first $\mathrm{cnt}(T_{1,1})$ nodes of S'',
  - Connect the next $n/2 - \mathrm{cnt}(T_{1,0})$ output nodes with the remaining nodes of S',
  - Connect the remaining output nodes with the remaining nodes in S''.

  (2) *Recursive call:* (see Figure 6).
  Independently in parallel:
  - Apply recursively the algorithm to S', the first $n/2$ input nodes and the subtree of the distribution tree with the root in $T_{1,0}$.
  - Apply recursively the algorithm to S'', the last $n/2$ input nodes and the subtree of the distribution tree with the root in $T_{1,1}$.

  (3) Output the composition of networks constructed in (1) and (2).

One can easily verify by induction on n that the network constructed above defines matching μ properly. Moreover, it is easy to see that given a distribution tree, the network is determined in a unique way.

In order to generate the network described we perform the distribution phase and the recursive call in parallel. It is possible, since the connections generated by both parts of the algorithm depend on the distribution tree $\langle \mathcal{T}, \mathrm{cnt} \rangle$ which is given. So the construction can be done by a CREW PRAM in constant time.

3.3 Random choice of a distribution tree

As we have seen, in order to choose a splitter it suffices to choose a distribution tree. A straightforward way would be to construct the tree top-down as follows.

Algorithm 3 *Naive method*

  $\mathrm{cnt}(T_{0,0}) \leftarrow \frac{n}{2}$
  `for` $i = 0$ `to` $\log n - 1$ `do`
    `for` $0 \le j < 2^i$ `do in parallel`
      choose $\mathrm{cnt}(T_{i+1,2j})$ as a random number according to probability
          distribution $\mathrm{Hp}(\mathrm{cnt}(T_{i,j}), \frac{n}{2^i})$
     $\mathrm{cnt}(T_{i+1,2j+1}) \leftarrow \mathrm{cnt}(T_{i,j}) - \mathrm{cnt}(T_{i+1,2j})$
    `endfor`

Given a value $\mathrm{cnt}(T_{i,j})$ we choose the values $\mathrm{cnt}(T_{i+1,2j})$ and $\mathrm{cnt}(T_{i+1,2j+1})$ so that $\mathrm{cnt}(T_{i+1,2j}) + \mathrm{cnt}(T_{i+1,2j+1}) = \mathrm{cnt}(T_{i,j})$. Given that we have already set

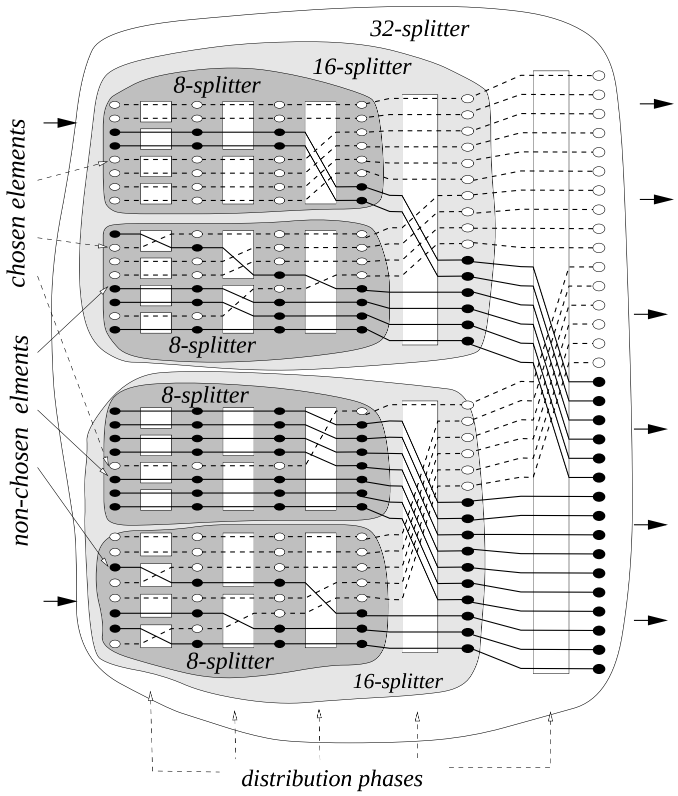

Fig. 6. The recursive structure of a splitter

$\mathrm{cnt}(T_{i,j}) = l$, then we know that we choose l nodes in $\mathcal{P}_{i,j}$. Thus we set $\mathrm{cnt}(T_{i+1,2j}) = k, \mathrm{cnt}(T_{i+1,2j+1}) = l - k$ with probability

$$\binom{n/2^{i+1}}{k} \cdot \binom{n/2^{i+1}}{l-k} \Big/ \binom{n/2^i}{l},$$

that is, according to the hypergeometric probability distribution $\mathrm{Hp}(l, n/2^{i+1})$.

We prove by induction on n that with this construction, when we start with $\mathrm{cnt}(T_{0,0}) = l$, each subset of l nodes is chosen by the distribution tree equiprobably. For $n = 1$ this is obviously true. Let us assume that the claim holds for $n/2$; we check it for n. Consider a set X of l out of n nodes such that $|X \cap \mathcal{P}_{1,0}| = k$. In order to choose X we have to decide upon $\mathrm{cnt}(T_{1,0}) = k$, $\mathrm{cnt}(T_{1,1}) = l - k$. This happens with probability $\binom{n/2}{k} \cdot \binom{n/2}{l-k} \cdot \binom{n}{l}^{-1}$. Function cnt is defined for the ancestors of $T_{1,0}$ so that, by the induction hypothesis, each subset of $\mathcal{P}_{1,0}$ of k elements is chosen with the same probability $\binom{n/2}{k}^{-1}$. Similarly, each subset of $\mathcal{P}_{1,1}$ of $l - k$ elements is chosen with probability $\binom{n/2}{l-k}^{-1}$. The processes of constructing subtrees with the roots $T_{1,0}$ and $T_{1,1}$ are independent, so the probability that we get X equals

$$\binom{n/2}{k} \cdot \binom{n/2}{l-k} \cdot \binom{n}{l}^{-1} \cdot \binom{n/2}{k}^{-1} \cdot \binom{n/2}{l-k}^{-1} = \binom{n}{l}^{-1},$$

as required.

Unfortunately, Algorithm 3 runs in $O(\log n)$ time. This follows from the fact that to generate $\mathrm{cnt}(T_{i+1,2j})$ and $\mathrm{cnt}(T_{i+1,2j+1})$ we have to wait until $\mathrm{cnt}(T_{i,j})$ is fixed. In the following we show how to avoid this delay.

3.4 Fast parallel generation of a distribution tree

In order to speed up the naive algorithm we apply the following trick. We substitute each $T_{i,j}$ by a set $\{T_{i,j,a} \mid 0 \le a \le \frac{n}{2^i}\}$ and by $T_{i,j,a}$ we understand a copy of $T_{i,j}$, which assumes that $\mathrm{cnt}(T_{i,j}) = a$. In other words, $T_{i,j,a}$ presumes that a nodes are to be chosen from $\mathcal{P}_{i,j}$. Let $\mathcal{T}' = \{T_{0,0,\frac{n}{2}}\} \cup \bigcup_{i=1}^{\log n} \bigcup_{j=0}^{2^i-1} \{T_{i,j,a} \mid 0 \le a \le \frac{n}{2^i}\}$. The following algorithm lets each $T_{i,j,a}$ choose at random its children $T_{i+1,2j,a'}$ and $T_{i+1,2j+1,a''}$ so that $a = a' + a''$.

Algorithm 4 *Generating step*

> `for each` vertex $\tau = T_{i,j,a} \in \mathcal{T}'$ where $i < \log n$
> `do in parallel`
>   choose a random number r according to probability distribution $\mathrm{Hp}(a, \frac{n}{2^i})$
> $\mathrm{Child}_{\mathrm{left}}(\tau) \leftarrow T_{i+1,2j,r}$
> $\mathrm{Child}_{\mathrm{right}}(\tau) \leftarrow T_{i+1,2j+1,a-r}$

Since $|\mathcal{T}'| = 1 + 2 \cdot \frac{n}{2} + 4 \cdot \frac{n}{4} + \ldots = O(n \log n)$, the generating step can be performed in a constant time with $O(n \log n)$ processors.

The generating step constructs a graph $\mathcal{G}$ with the set of vertices $\mathcal{T}'$ and the edges $(t, \mathrm{Child}_{\mathrm{left}}(t))$ and $(t, \mathrm{Child}_{\mathrm{right}}(t))$. It is easy to see that the successors of $T_{0,0,\frac{n}{2}}$ in $\mathcal{T}'$ form a complete binary tree of depth $\log n$. Let us call this tree $\overline{\mathcal{T}'}$ and let us define $\overline{\mathrm{cnt}}$ for each vertex of this tree to be $\overline{\mathrm{cnt}}(T_{i,j,a}) = a$. Then $\langle \overline{\mathcal{T}'}, \overline{\mathrm{cnt}} \rangle$ is a well defined distribution tree generated according to the probability distribution defined in the previous subsection.

In order to use tree $\overline{\mathcal{T}'}$ in our construction of a splitter in Algorithm 2 we still lack one important thing. We do not know how to find the vertices of $\overline{\mathcal{T}'}$ without tracing $\overline{\mathcal{T}'}$ from the root. The next algorithm gathers information about all vertices of $\overline{\mathcal{T}'}$ in the root of $\overline{\mathcal{T}'}$ (and for technical reasons, at some vertices of $\overline{\mathcal{T}'}$ as long as it is needed.) The technique that we use is a kind of pointer jumping. To collect information about its subtree each vertex τ of $\mathcal{G}$ constructs a local binary tree *InfoTree*(τ) (stored at some known locations). Initially, each *InfoTree*(τ) stores the names of the children of τ. Through "pointer jumping" it is possible to replace leaves of *InfoTree*(τ) by subtrees found by the vertices corresponding to the leaves of *InfoTree*(τ):

Algorithm 5 *Gathering step*

> `for each` vertex $t \in \mathcal{T}'$ `do in parallel`
>   set *InfoTree*(t) to be the tree with root t and the leaves $\mathrm{Child}_{\mathrm{left}}(t), \mathrm{Child}_{\mathrm{right}}(t)$
> `repeat` $\log\log n$ times
>   `for each` vertex $t \in \mathcal{T}'$ `do in parallel`
>   `if` $l_1, l_2, \ldots, l_r$ are the (names of the) leaves of *InfoTree*(t),
>   `then` for $1 \le j \le r$ replace the leaf l_j in *InfoTree*(t) by *InfoTree*(l_j)

Lemma 7. *Algorithm 5 collects the whole tree $\overline{\mathcal{T}'}$ in InfoTree$(T_{0,0,\frac{n}{2}})$ in $O(\log\log n)$ time using $O(n^2)$ processors.*

Proof. The repeat loop of the algorithm is executed in a constant time – simply for each leaf and each value copied there is a separate processor. Therefore the runtime is $O(\log\log n)$. The number of the processors used for the kth iteration does not exceed the total size of all *InfoTree*'s immediately after iteration k. Notice that for each $T_{i,j,a} \in \mathcal{T}'$ the *InfoTree* consists finally of $2^{\log n - i}$ vertices. Thus the total number of processors used at any iteration is at most

$$\sum_{i=0}^{\log n} \sum_{j=0}^{2^i-1} \sum_{a=0}^{n/2^i} 2^{\log n - i} = \sum_{i=0}^{\log n} 2^i \cdot (\frac{n}{2^i} + 1) \cdot 2^{\log n - i} \leq n(\log n + 1) + 2n^2.$$

By applying the generating step and then the gathering step we would get an algorithm which generates a distribution tree in time $O(\log\log n)$ using $O(n^2)$ processors. In the following we show how to combine the gathering step with Algorithm 3 in order to reduce the number of processors without loosing the execution speed.

Algorithm 6 *Fast parallel method for generating and identification of a distribution tree; parameter h $(0 < h < \log n)$ will be used to tune the execution time and the number of processors used:*

(1) execute the generating step
(2) `for each` $\tau \in \mathcal{T}'$ `do in parallel`
  perform $\log h$ times the loop of the gathering step, so that *InfoTree*(τ) will contain a tree of depth h pointing to all successors of τ in distance at most h
(3) inform $T_{0,0,\frac{n}{2}}$ that it belongs to $\overline{\mathcal{T}'}$
(4) `for` $i \leftarrow 1$ `to` $\log n$ `step by` h
    `for each` τ, a vertex of $\mathcal{T}'$ of depth i `do in parallel`
      `if` τ knows that it belongs to $\overline{\mathcal{T}'}$ `then`
        τ informs all its successors pointed by the vertices of *InfoTree*(τ),
        that they belong to $\overline{\mathcal{T}'}$
(5) `for` each i, j, a
    `if` vertex $T_{i,j,a}$ is marked as chosen (and thereby is in $\overline{\mathcal{T}'}$),
    `then` set $\mathrm{cnt}(T_{i,j}) = a$

Lemma 8. *For $h = \frac{\log n}{\log\log n}$, Algorithm 6 runs in time $O(\log\log n)$ and uses $O(n^{1+1/\log\log n} \cdot \log n)$ processors.*

Proof. Let us consider each phase of Algorithm 6 separately:

Phase 1: By Proposition 7 the generating step takes constant time with $O(n \log n)$ processors.

Phase 2: Each execution of the loop of the gathering step is performed in a constant time. So together $\log h$ executions of the loop take $O(\log h)$ time. As during this step each $\tau \in \mathcal{T}'$ collects only information about its successors at distance $\leq h$, *InfoTree*'s have depths at most h and thereby at most 2^h leaves each. So the total number of processors necessary to perform this step is

$$\sum_{i=0}^{\log n}\sum_{j=0}^{2^i-1}\sum_{a=0}^{\frac{n}{2^i}} 2^h = 2^h \cdot \sum_{i=0}^{\log n} 2^i(\frac{n}{2^i}+1) \leq 2 \cdot 2^h n \log n.$$

Phase 4: The loop is executed $\frac{\log n}{h}$ times and its body is performed in one parallel step. The number of processors used does not exceed the number of vertices in $\overline{\mathcal{T}'}$, that is $O(n \log n)$.

So finally, for $h = \frac{\log n}{\log\log n}$, the total running time of Algorithm 6 is bounded by

$$O(1) + O(\log h) + O(\frac{\log n}{h}) = O(\log\log n).$$

The number of processors used is

$$O(n \log n) + O(2^h n \log n) = O(n^{1/\log\log n} n \log n).$$

Concluding we get the following result:

Corollary 9. *The distribution tree can be generated in time $O(\log\log n)$ using $O(n^{1+1/\log\log n} \cdot \log n)$ processors.*

3.5 Properties of the constructions

It is easy to see that the splitter constructed using Algorithm 2 has depth $\log n$. Hence applying it in Algorithm 1 we get a random matching network $\mathcal{N}$ with $O(\log^2 n)$ levels. By Corollary 9, generating $\mathcal{N}$ takes $O(\log\log n)$ time and uses $O(n^{1+1/\log\log n} \cdot \log n)$ processors. Performing pointer jumping on $\mathcal{N}$ takes $O(\log\log^2 n) = O(\log\log n)$ steps and uses $O(n \log n)$ processors. Thereby the algorithm designed fulfills the properties stated in Theorem 3.

4 Conclusions

Our CREW algorithm uses hypergeometric random number generator. It would be desirable to achieve the same running time using only the unbiased coins. However, it is easy to see that then the probability of generating an arbitrary permutation would be of the form $\frac{i}{2^j}$, for some $i, j \in \mathbf{N}$. So it cannot be $\frac{1}{n!}$ and therefore each strong randomized algorithm has to use a nonuniform number generator. It would be interesting to find an $o(\log n)$-time strong randomized algorithm based on a simple number generator or to provide an elegant parallel generator for hypergeometric distribution.

References

1. R. Anderson, Parallel algorithms for generating random permutations on a shared memory machine, in *Proc. 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, (ACM Press, New York, 1990), 95–102.
2. A. Borodin and J. E. Hopcroft, Routing, merging, and sorting on parallel models of computation, *J. Comput. System Sci.* **30** (1985), 130–145.
3. R. Cole, Parallel merge sort, *SIAM J. Comput.* **17** (1988) 770–785.
4. S. Cook, C. Dwork, and R. Reischuk, Upper and lower bounds for parallel random access machines without simultaneous writes, *SIAM J. Comput.* **15** (1986) 87–97.
5. M. Dietzfelbinger, M. Kutyłowski, and R. Reischuk, Exact lower time bounds for computing boolean functions on CREW PRAMs, *J. Comput. System Sci.* **48** (1994) 231–254.
6. R. Durstenfeld, Random permutation (Algorithm 235), *Comm. ACM* **7** (1964) 420.
7. A. Gottlieb and C. P. Kruskal, Complexity results for permuting data and other computations on parallel processors, *J. Assoc. Comput. Mach.* **31** (1984) 193–209.
8. T. Hagerup, Fast parallel generation of random permutations, in *Proc. 18th Annual International Colloquium on Automata, Languages and Programming*, ICALP'91, (Springer Verlag, LNCS 510, Heidelberg, 1991), 405–416.
9. D. E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*, volume 2, Addison-Wesley, Reading, Massachusetts, 2nd edition, 1981.
10. Y. Matias and U. Vishkin, Converting high probability into nearly-constant time - with applications to parallel hashing, in *Proc. 23rd Annual ACM Symposium on Theory of Computing*, (ACM Press, New York, 1991), 307–316.
11. G. L. Miller and J. H. Reif, Parallel tree contraction, in *Proc. 26 Symposium on Foundations of Computer Science*, (IEEE, Los Alamitos, 1985), 478–489.
12. G. L. Miller and J. H. Reif, Parallel tree contraction, Part 1: Fundamentals, *Adv. in Comput. Res.* **5** (1989) 47–72.
13. R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.
14. K. Mulmuley, *Computational Geometry. An Introduction Through Randomized Algorithms*, Prentice Hall, Englewood Cliffs, New Jersey, 1994.
15. S. Rajasekaran and J. Reif, Optimal and sublogarithmic time randomized parallel sorting algorithms, *SIAM J. Comput.* **19** (1989) 594–607.
16. J. Reif, An optimal parallel algorithm for integer sorting, in *Proc. 26 Symposium on Foundations of Computer Science*, (IEEE, Los Alamitos, 1985), 490–503.
17. R. Sedgewick, Permutation generation methods, *ACM Comput. Surv.* **9** (1977) 138–164.

This article was processed using the LaTeX macro package with LLNCS style