Fast Generation of Random Permutations via Networks Simulation*†

Artur Czumaj‡ Przemysława Kanarek§

Mirosław Kutyłowski‡ Krzysztof Loryś¶

Abstract

We consider the 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 outperforms each of the previously known algorithms for the exclusive write PRAMs.

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

Keywords: parallel algorithms, random permutation, uniform distribution, switching networks, matching, PRAM, CREW, EREW.

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

†A preliminary version appeared in *Proceedings of the 4th Annual European Symposium on Algorithms* (ESA'96), volume 1136 of *Lecture Notes in Computer Science*, pages 246–260, Springer-Verlag, Berlin, 1996.

‡Heinz Nixdorf Institute and Department of Mathematics & Computer Science, University of Paderborn, D-33095 Paderborn, Germany, E-mail: artur,mirekk@uni-paderborn.de.

§Institute of Computer Science, University of Wrocław, Przesmyckiego 20, PL-51-151 Wrocław, Poland, E-mail: pka@tcs.uni.wroc.pl.

¶Department of Computer Science, University of Trier, D-54286 Trier, Germany, and Institute of Computer Science, University of Wrocław, Przesmyckiego 20, PL-51-151 Wrocław, Poland, E-mail: lorys@tcs.uni.wroc.pl.

1 Introduction

Generating permutations is a fundamental problem studied in theoretical and applied computer science already for few decades. One approach (historically first) assumed generation of all permutations of n elements (see [8] and [17], and the references therein). Evidently, the problem is intractable even for very moderate values of n. The next approach, also extensively investigated, is generating permutations at random. This task can be formally described as follows:

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.

The problem of generating a random permutation has recently received a lot of attention. The reasons are manifold. One of them is the growing interest in randomized algorithms (see e.g. [12, 13]). 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. Another field in which random permutations might be important is cryptography. Random objects such as permutations are components of a large number of cryptographic algorithms and protocols. Having really random fast and cheap source of these objects would be crucial for avoiding security gaps and is often assumed by analysis of cryptographic protocols. Unfortunately, creating such sources is a challenging problem and currently used techniques generate only pseudorandom objects (see e.g. [14]). One of many reasons why random permutations are difficult to generate is the running time involved that is unacceptable for practical applications. We believe that parallel techniques might be important in this context.

When we consider generation of a random permutation, we have to assume that our algorithm uses some random resources. Thereby the algorithm must be randomized. However, there are many ways of using randomness and a sentence *algorithm A generates a random permutation in time T* may have many different meanings. We are interested in algorithms where all bounds are guaranteed in a strong way:

Definition 2 We say that a machine M generating a random permutation of n elements in time T with p processors and m memory cells is *strong randomized*, if:

- M always halts after T steps,
- M may use given p processors and m cells,
- each permutation has equal chance of $1/n!$ to become the output of M.

In the literature there are many algorithms for generating random permutations that do not satisfy these conditions. In this case we say that the algorithm is *weak randomized*. This happens if, for instance, an algorithm ensures the claimed time bound to hold only in the expected case, or to hold with high probability. For example, a randomized algorithm may generate a permutation uniformly at random and halt within T steps provided that a certain (randomized) event $\mathcal{E}$ takes place, where $\Pr(\mathcal{E}) > 1 - o(1)$ and $\mathcal{E}$ is independent of the permutation found.

There are many reasons why permutation generation by strong randomized algorithms are superior to weak randomized ones. For instance, it may be difficult to check that the mentioned event $\mathcal{E}$ in the case of a weak algorithm has really occurred. In this case we cannot guarantee that the algorithm gives proper answers. 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 in this situation the probability of a single permutation may differ from the ideal $1/n!$ even by $f(n)$. (Note that $f(n)$ is usually an extremely big value compared with $1/n!$.) An approach of this kind may be 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.

In this paper we study the permutation generation problem on the Parallel Random Access Machine (PRAM) model. We focus on PRAMs with exclusive read exclusive write (EREW) and concurrent read exclusive write (CREW) access mode to the shared memory. In order to make our model sound we assume that a cell of an n-processor PRAM may store $O(\log n)$ bits.

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 [10, 11, 15] culminating in the papers due to Hagerup [7] and Matias and

Vishkin [9], 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* and it also leads to *weak* randomized algorithms. 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 a 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 [7] used 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 [8, 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 a parallel setting and showed that it can be run efficiently on parallel machines with a small number of processors communicating through a common bus such as Ethernet. 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 delivery times of messages sent through the bus are determined on-line by an adversary controlling the bus. Hagerup [7] later implemented SHUFFLE to run in $O(\log n)$ time with n processors and $\Theta(n^2)$ 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

fastest strong randomized $O(\log n)$-time algorithm uses n-processors and $\Omega(n^2)$ space on the EREW PRAM. We are not aware of any better strong randomized algorithm existing in the literature even on the CRCW PRAM !

1.2 New results

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

Theorem 1 *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 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. Though the algorithm is simple, it improves upon 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 the algorithm strong randomized and remove concurrent reads, while the other parameters are not worsened.

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

Theorem 2 *There is a strong randomized CREW PRAM algorithm that generates permutations of n elements uniformly at random in time $O(\log\log n)$ using $O(n^{1+\frac{1}{c\log\log n}})$ processors, for arbitrary positive constant c.*

The main message of this result is that the generation of random permutations may follow another strategy than the previously known techniques and lead to possible more efficient algorithms.

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 $i/2^j$, for some $i, j \in \mathbf{N}$. So it cannot be $1/n!$ and therefore each strong randomized algorithm has to use something more than a uniform random bit generator.

While designing our algorithms we introduce a novel technique for generating permutations that we call *networks simulation*. 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

It is equivalent to construct a random permutation on n elements or to construct a random perfect matching between two sets $\{a_1, \ldots, a_n\}$ and $\{b_1, \ldots, b_n\}$ of n elements each. Simply, if μ is such a perfect matching, then we define $\pi_\mu(i) = j$ if $\mu(a_i) = b_j$. Therefore throughout the rest of the paper we may talk about perfect matchings instead of permutations.

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 choosing the matchings between the levels must be carefully done 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 every node R in level j stores a pointer to the single node S on level $\min\{j + 2^i, \texttt{last level}\}$ 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 of R points to a node S' on level $\min\{j + 2^{i+1}, \texttt{last level}\}$ such that there is a path between R and S'.

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

2 EREW Algorithm

In this section we prove Theorem 1. The algorithm SHUFFLE can be implemented by constructing a so called *shuffle network* with n levels. The matching between levels i and $i+1$ contains a single switch (i, k_i), where k_i, $i \leq k_i \leq n$, is chosen uniformly at random and corresponds to the number κ chosen during the ith iteration of the loop of SHUFFLE. The switch (i, k_i) connects the node i from level i with node k_i from level $i+1$, and node k_i from level i with node i from level $i+1$. For $j \neq i, k_i$, the node j of level i is connected with node j of level $i+1$. For examples see Figure 1.

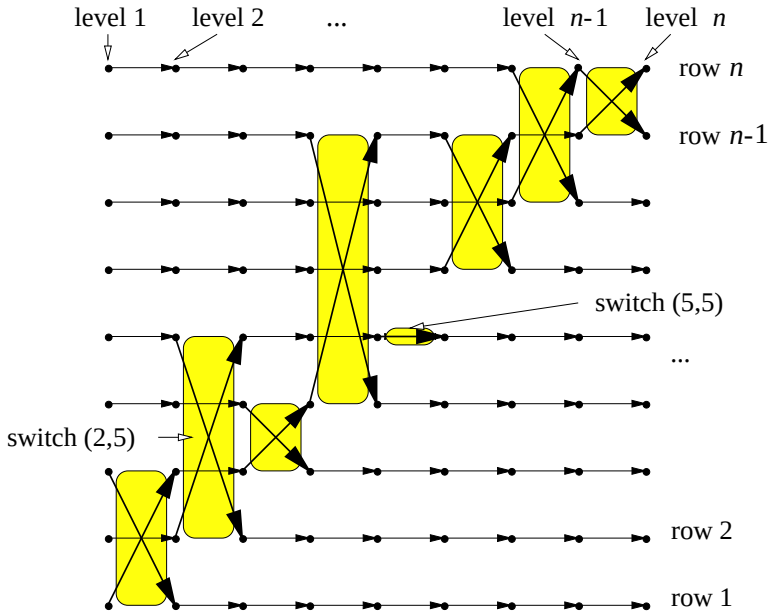

Figure 1: A shuffle network. Edges corresponding to the switches are distinguished by the shadowed fields.

To find the permutation defined by a shuffle network we may apply pointer jumping, but it would require roughly 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.

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 (see Figure 2). Our goal is to find the final node of each path v_i. Note that each path v_i contains three types of edges. First two types correspond to edges of switches. An edge of a switch 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*. More precisely, v_i has a *horizontal subpath* 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 $k_j = l$. Note that the number of horizontal subpaths is $O(n)$. Indeed, each horizontal subpath starts either at the first level or at the node of a switch. There are n switches and each gives rise to at most two horizontal subpaths. Our construction is based on contracting horizontal subpaths to a single edges.

In order to find the final nodes of the paths, we generate a directed graph G

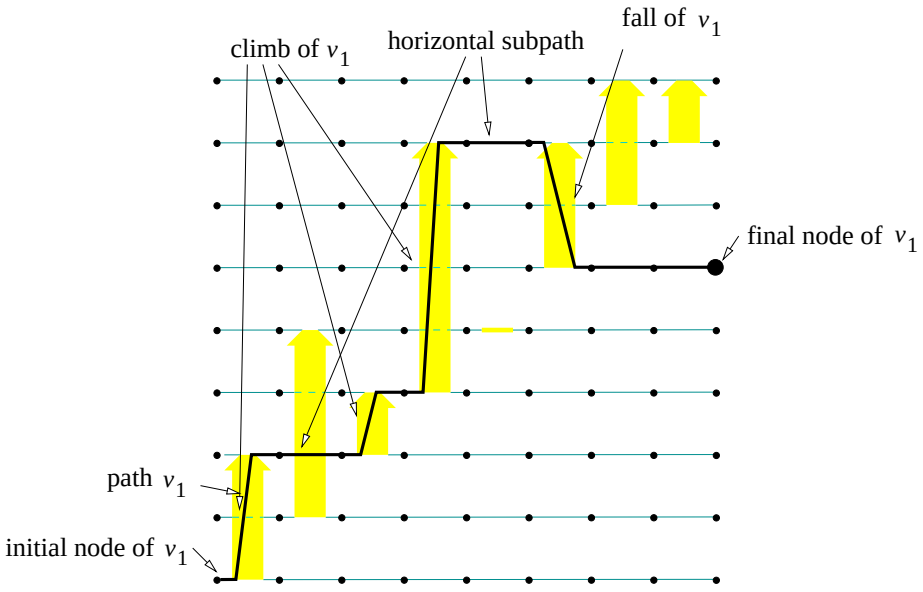

Figure 2: Path v_1 in a shuffle network

consisting of $O(n)$ nodes and $O(n)$ edges (see Figure 3). There are n nodes that correspond to the starting positions at level 1, and we denote them by $\langle i, 0, i\rangle$, for $1 \leq i \leq n$. The remaining nodes correspond to the switches: two nodes per switch. The two nodes corresponding to a switch (i, k_i), $1 \leq i \leq n$, are denoted by $\langle i, i, k_i\rangle$ and $\langle k_i, i, k_i\rangle$ (the first coordinate is the row number, the next two coordinates correspond to the switch.) There are also n "output" nodes out(i), for $1 \leq i \leq n$.

There are three kinds of edges in G. Some edges correspond to the edges in the shuffle network where the paths climb. Thus for each i, $1 \leq i \leq n$, if $k_i \neq i$, then there is a directed edge from $\langle i, i, k_i\rangle$ to $\langle k_i, i, k_i\rangle$. There are $O(n)$ edges corresponding to the horizontal subpaths. We take a directed edge from $\langle l, j, l\rangle$ to $\langle l, l, k_l\rangle$, if a path arriving at row l through switch (j, l) leaves row l through switch (l, k_l), where $k_l > l$ (that is, the path further climbs). An important point is that we may find these edges by lexicographical sorting triplets $\langle s, i, j\rangle$. The key observation is that if $\langle j, j, k\rangle$ is the immediate successor of $\langle j, s, j\rangle$ after lexicographical sorting, then there is a horizontal subpath between levels $s + 1$ and j inside row j. Using the sorting algorithm due to Cole [3], sorting the triplets can be performed in $O(\log n)$ time on an n processor EREW PRAM with $O(n)$ memory cells.

There are also n edges corresponding to the paths starting with falling edges

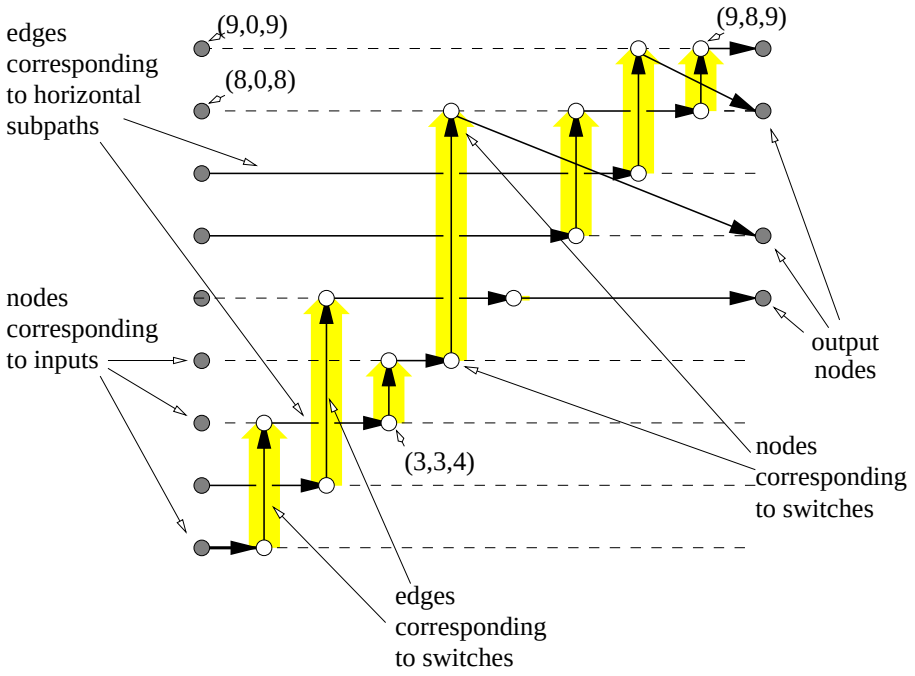

Figure 3: Edges in graph G corresponding to the paths v_1, v_2, v_6, v_7

and leading to the output nodes. We find them in the following way: If after sorting the triplets $\langle s, i, s\rangle$ is followed by $\langle s, j, s\rangle$ for some $j \leq s$, then G contains an edge from $\langle s, i, s\rangle$ to out(j).

It follows from the construction that the edges in G determine the paths corresponding to the paths v_i: If the last node of a path starting in $\langle s, 0, s\rangle$ is out(j), then j is the final node of v_s. 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, each path in G has length at most n and pointer jumping takes $O(\log n)$ time. □

Observe that our construction reduces permutation generation to stable integer sorting, or alternatively, to sorting distinct integers drawn from the set $\{1, \ldots, n^3\}$. 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 of Hagerup [7].

3 CREW Algorithm

This section contains a proof of Theorem 2. In our construction we use the following probability distribution.

Definition 3 We say that a random variable $X \in \{0, 1, \ldots, l\}$ has *hypergeometric probability distribution* $\mathrm{Hp}(l, m)$ if

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

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

Our CREW PRAM algorithm requires that each processor can choose randomly an integer from the set $\{0, 1, \ldots, m\}$, $m \leq n$, 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 4):

Definition 4

1. 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 $\mathcal{B}$.

2. Let l be an even integer. An *l-splitter* between sequences of nodes $\mathcal{P} = P_0, \ldots, P_{l-1}$ and $\mathcal{R} = R_0, \ldots, R_{l-1}$ is a network which for some increasing sequence of integers $i_0, \ldots, i_{l/2-1}$, $0 \leq i_0$, $i_{l/2-1} < l$,

   - defines a stable perfect matching between nodes $P_{i_0}, \ldots, P_{i_{l/2-1}}$ and nodes $R_0, \ldots, R_{l/2-1}$,
   - defines a stable perfect matching between the sequences 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*. The nodes in $\mathcal{P}$ are called *input nodes* and the nodes in $\mathcal{R}$ are called *output nodes*.

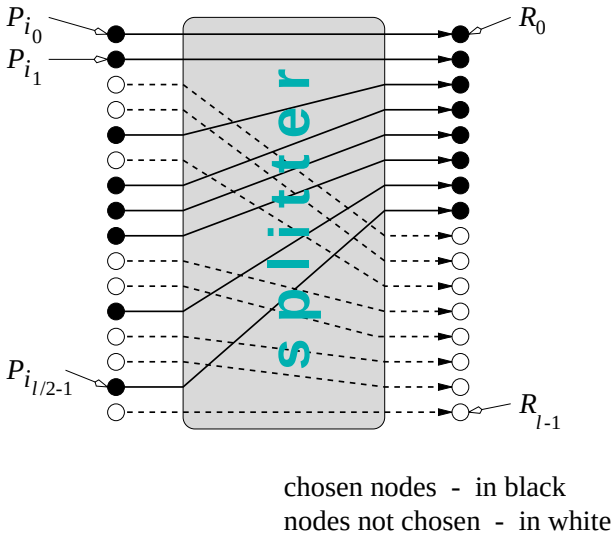

Figure 4: The matching defined by a stable splitter

In the next subsection we show how to construct a random splitter. "Random" means in this context that for a given l and a sequence $\mathcal{P}'$ 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 sequences of nodes* $\mathcal{P} = P_0, \ldots, P_{m-1}$ *and* $\mathcal{R} = R_0, \ldots, R_{m-1}$ *(see Figure 5.)*

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

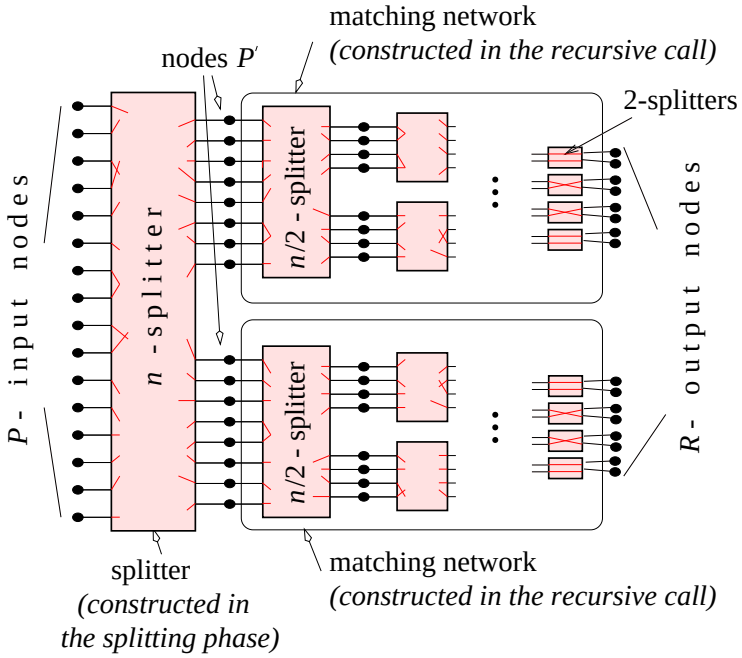

Figure 5: Structure of the matching network constructed by Algorithm 1

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.

Proposition 3.1 *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 obtain 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 obtain 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, $0 \leq i \leq \log n$, let the sets $\mathcal{P}_{i,j}$, $0 \leq j < 2^i$, partition the set of input nodes $\mathcal{P} = \{P_0, P_1, \ldots, P_{n-1}\}$ into 2^i consecutive intervals of length $n/2^i$. That is, we put $\mathcal{P}_{i,j} = \{P_{j \cdot n/2^i}, \ldots, P_{(j+1) \cdot n/2^i - 1}\}$. According to this definition each set $\mathcal{P}_{\log n, j}$ consists of a single input P_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 $\{T_{i,j} \mid 0 \leq i \leq \log n \text{ and } 0 \leq j < 2^i\}$. 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:

- $\mathrm{cnt}(T_{0,0}) = n/2$,
- $\mathrm{cnt}(T_{i,j}) = \mathrm{cnt}(T_{i+1,2j}) + \mathrm{cnt}(T_{i+1,2j+1})$,
- $0 \leq \mathrm{cnt}(T_{i,j}) \leq |\mathcal{P}_{i,j}|$,

for $0 \leq i < \log n$ and $0 \leq j \leq 2^i$. 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 $\mathrm{cnt}(T_{\log n, j}) = 1$. For an example, see Figure 6.

How to construct uniformly at random a distribution tree without loosing efficiency will be discussed in the next subsections. At this moment we assume

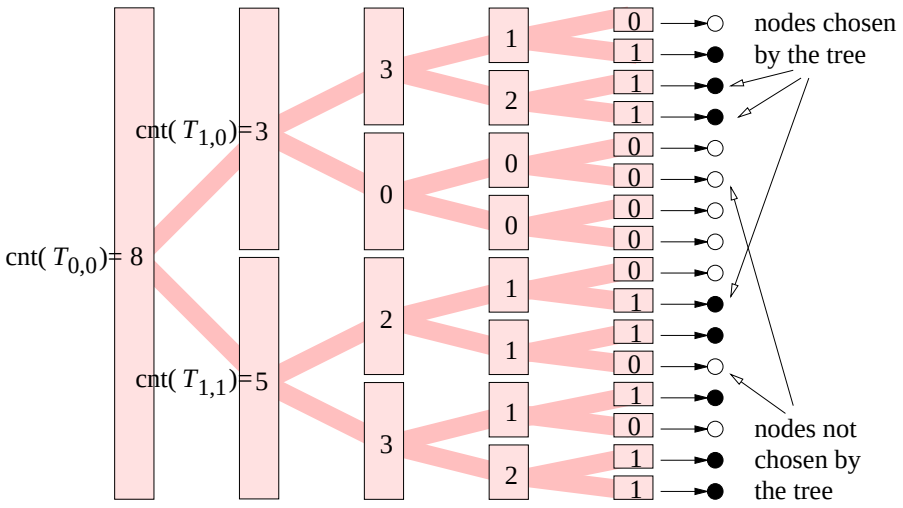

Figure 6: A distribution tree

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

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

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 from distribution tree* $\langle \mathcal{T}, \mathrm{cnt} \rangle$

`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 $S' = \{P'_0, \ldots, P'_{n/2-1}\}$ and $S'' = \{P'_{n/2}, \ldots, P'_{n-1}\}$.

(1) *Recursive call:* (see Figure 7)
Independently do 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 $T_{1,1}$.

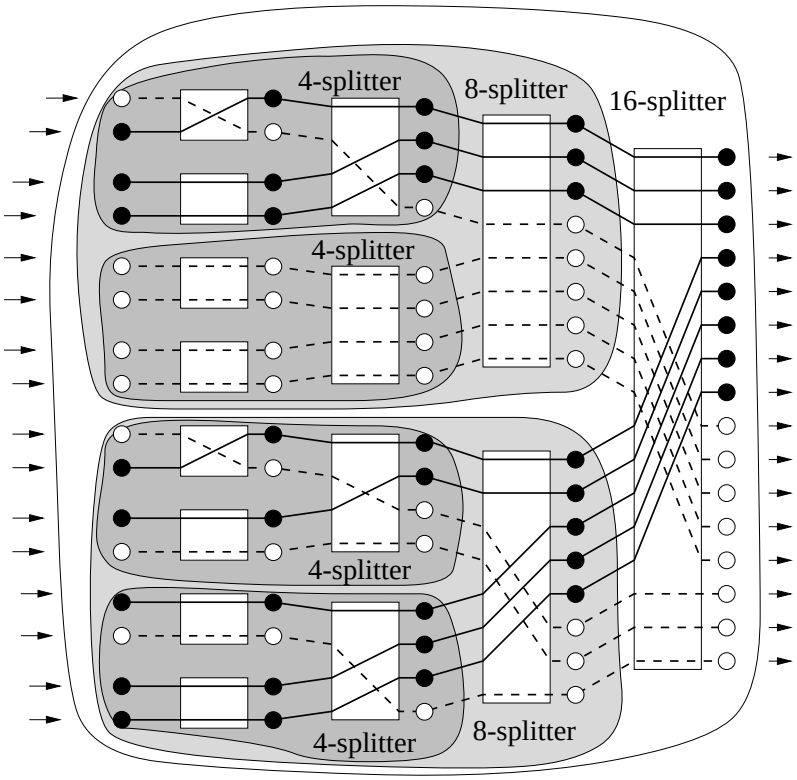

Figure 7: The recursive structure of a splitter

(2) *Distribution phase:* (see Figure 8)

- connect the first $\mathrm{cnt}(T_{1,0})$ nodes of S' with the first $\mathrm{cnt}(T_{1,0})$ output nodes,
- connect the first $\mathrm{cnt}(T_{1,1})$ nodes of S'' with the next $\mathrm{cnt}(T_{1,1})$ output nodes,
- connect the remaining nodes of S' with the next $n/2 - \mathrm{cnt}(T_{1,0})$ output nodes,
- connect the remaining nodes in S'' with the remaining output nodes the remaining nodes in S''.

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

We prove by induction on n that the network constructed above defines matching μ. For $n = 1$ it is obvious, so let us assume that $n > 1$. By the construction, for $i \leq \mathrm{cnt}(T_{1,0})$ the ith chosen input node is connected with

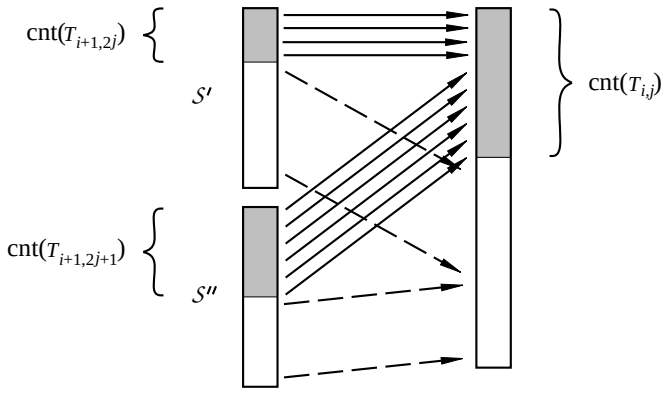

Figure 8: Matching constructed by the distribution phase

the ith node of S' and therefore with the ith output node. For $i > \mathrm{cnt}(T_{1,0})$, the ith chosen node is the $i - \mathrm{cnt}(T_{1,0})$ chosen node in the second half of the set of input nodes. Hence it is connected with the node $i - \mathrm{cnt}(T_{1,0})$ of S''. So this input node is finally connected with the output node with the index $i - \mathrm{cnt}(T_{1,0}) + \mathrm{cnt}(T_{1,0}) = i$. We conclude that the chosen nodes are matched with the output nodes indicated by μ. Similarly we may check that the non-chosen nodes are connected to the output nodes according to μ, too.

In order to generate the splitter described we perform the distribution phase and the recursive call in parallel. Because the connections generated by both parts of the algorithm depend on the distribution tree $\langle \mathcal{T}, \mathrm{cnt} \rangle$ that is given in advance, 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 uniformly at random it suffices to choose a distribution tree uniformly at random. A straightforward way would be to construct the tree top-down as follows.

Algorithm 3 *Naive method for constructing random distribution tree*

$\mathrm{cnt}(T_{0,0}) \leftarrow n/2$
`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 $\mathrm{Hp}(\mathrm{cnt}(T_{i,j}), n/2^i)$
    $\mathrm{cnt}(T_{i+1,2j+1}) \leftarrow \mathrm{cnt}(T_{i,j}) - \mathrm{cnt}(T_{i+1,2j})$
  `endfor`

Let us have a closer look at the above algorithm. 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 $\mathrm{cnt}(T_{i+1,2j}) + \mathrm{cnt}(T_{i+1,2j+1}) = \mathrm{cnt}(T_{i,j})$, as demanded for function cnt. Given that we have already set $\mathrm{cnt}(T_{i,j}) = l$, then we know that we choose l nodes in $\mathcal{P}_{i,j}$. Thus we should choose $\mathrm{cnt}(T_{i+1,2j}) = k, \mathrm{cnt}(T_{i+1,2j+1}) = l-k$ (that is, k chosen elements from the first half and $l-k$ from the second half) with probability

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

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

Proposition 3.2 *When Algorithm 3 starts with* $\mathrm{cnt}(T_{0,0}) = l$*, each subset of* l *nodes is chosen by the distribution tree equiprobably.*

Proof: The proof is by induction on n. 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 it is necessary 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 obtain X equals

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

as required. □

Notice that Algorithm 3 runs in $O(\log n)$ time and therefore cannot be used as a subroutine of our algorithm mentioned in Theorem 2. The reason for this running time is that in order 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 subsection we show how to elude this difficulty.

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 $\{T_{i,j,a} \mid 0 \le a \le n/2^i\}$. 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 $\mathcal{T}' = \{T_{0,0,\frac{n}{2}}\} \cup \{T_{i,j,a} \mid 1 \le i \le \log n, 0 \le j < 2^i, 0 \le a \le n/2^i\}$. 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 $a = a' + a''$.

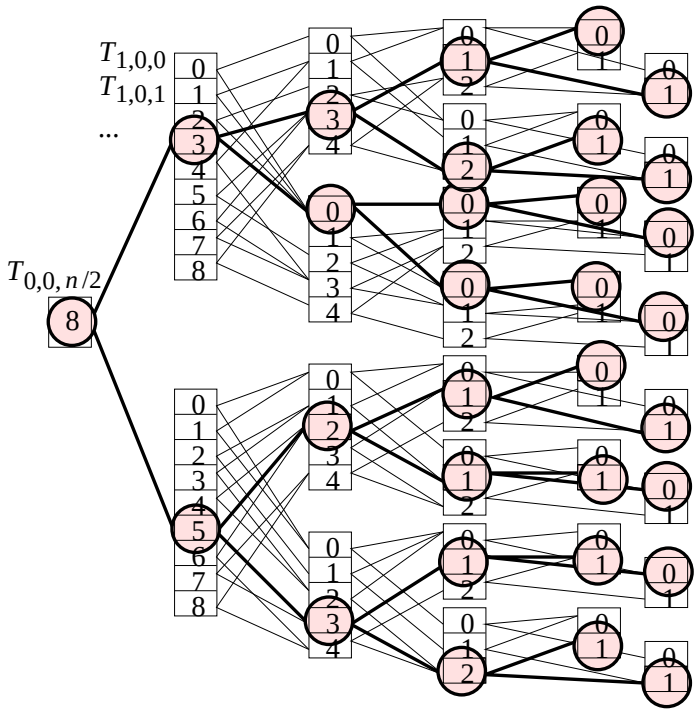

Figure 9: The edges chosen by the nodes of $\mathcal{T}'$. The bold edges correspond to the distribution tree generated by $\mathcal{T}'$

Algorithm 4 *Generating Step*

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

Since $|\mathcal{T}'| = 1 + 2 \cdot \frac{n}{2} + 4 \cdot \frac{n}{4} + \ldots = n \log n + 1$, Generating Step can be performed in a constant time with $n \log n$ processors.

Generating Step constructs a graph $\mathcal{G}$ with the set of vertices $\mathcal{T}'$ and the edges $(\tau, \mathrm{Child}_{\mathrm{left}}(\tau))$ and $(\tau, \mathrm{Child}_{\mathrm{right}}(\tau))$. It is easy to see that the successors of $T_{0,0,\frac{n}{2}}$ in $\mathcal{G}$ 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 $T_{i,j,a}$ of this tree to be 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. The distribution tree is given as a pointer structure

and we do not know how to find the vertices of $\overline{\mathcal{T}'}$ without tracing $\overline{\mathcal{T}'}$ from the root. On the other hand, for Algorithm 2 we need to find the labels of the nodes!

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.) For this purpose we use standard doubling technique. For each vertex τ of $\mathcal{G}$ we shall collect information about the subtree *InfoTree*(τ) rooted at τ and containing the successors of τ in $\mathcal{G}$. Initially, each *InfoTree*(τ) stores the names of the children of τ. Using the doubling technique it is possible to replace leaves of *InfoTree*(τ) by subtrees found up to this moment by the vertices corresponding to the leaves of *InfoTree*(τ):

Algorithm 5 *Gathering Step*

`for each` vertex $\tau = T_{i,j,a} \in \mathcal{T}'$ `do in parallel`
  `if` $i = \log n$ `then`
    set *InfoTree*(τ) to be the tree with only one vertex τ which is both the root and the leaf
  `if` $0 \leq i < \log n$ `then`
    set *InfoTree*(τ) to be the tree with root τ and the leaves $\mathrm{Child}_{\mathrm{left}}(\tau), \mathrm{Child}_{\mathrm{right}}(\tau)$

`repeat` $\log\log n$ times
  `for each` vertex $\tau \in \mathcal{T}'$ `do in parallel`
    `if` $l_1, l_2, \ldots, l_r$ are the leaves of *InfoTree*(τ),
    `then` for $1 \leq j \leq r$ replace the leaf l_j in *InfoTree*(τ) by *InfoTree*(l_j)

Proposition 3.3 *Gathering Step collects the whole tree* $\overline{\mathcal{T}'}$ *in* *InfoTree*($T_{0,0,\frac{n}{2}}$) *in* $O(\log\log n)$ *time using* 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 running time 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 $2 \cdot 2^{\log n - i} - 1$ vertices. Thus the total number of processors used at any iteration does not exceed

$$\begin{aligned} \sum_{T_{i,j,a} \in \mathcal{T}'} (2 \cdot 2^{\log n - i} - 1) &= \sum_{i=0}^{\log n} 2^i \cdot (\frac{n}{2^i} + 1) \cdot (2 \cdot 2^{\log n - i} - 1) \\ &= (2n-1)^2 + n \cdot (\log n + 1). \end{aligned}$$

□

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

The main idea behind our construction is to generate the *InfoTree*'s till some depth h is reached. If we stop generating *InfoTree*'s at the moment when they have depth $h \ll \log n$, then because the size of each tree is small, we use much less processors than during Gathering Step. Once this is done we shall determine the distribution tree $\langle \overline{\mathcal{T}'}, \overline{\mathrm{cnt}} \rangle$ sequentially top-down: We start with the root, then collect information about the nodes of the first h levels (using the information contained in *InfoTree* of the root), then about the nodes of the next h levels (using *InfoTree*'s of the nodes already *informed* at level $h+1$), and so on.

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 Generating Step

(2) `for each` $\tau \in \mathcal{T}'$ `do in parallel`
perform $\log h$ times the loop of 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` $i = 1, h+1, 2h+1, \ldots, \log n$ `do`
`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`
inform all successors of τ 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 $\mathrm{cnt}(T_{i,j}) = a$

Now let us analyze resources necessary to execute Algorithm 6.

Proposition 3.4 *For every integer* h, $1 \leq h \leq \log n$, *Algorithm 6 runs in time* $O(\log h + \log n/h)$ *and uses* $O(2^h n \log n)$ *processors.*

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

Phase 1: Generating Step takes constant time with $n \log n$ processors.

Phase 2: Each execution of the loop of 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 $2 \cdot 2^h - 1$ nodes each. So the total number of processors necessary to perform this step is bounded by

$$\begin{aligned} \sum_{T_{i,j,a} \in \mathcal{T}'} (2 \cdot 2^h - 1) &= \sum_{i=0}^{\log n} 2^i \cdot (\frac{n}{2^i} + 1) \cdot (2 \cdot 2^h - 1) \\ &\leq 3 \cdot 2^h \cdot n \log n. \end{aligned}$$

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, the total running time of Algorithm 6 is bounded by

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

The number of processors used is bounded by

$$\max\{n \log n, 3 \cdot 2^h \cdot n \log n\} = 3 \cdot 2^h \cdot n \log n$$

□

Concluding we get the following result:

Corollary 3.5 *For arbitrary positive constant c, the distribution tree can be generated in time $O(c \log\log n)$ using $n^{1+1/c \log\log n}$ processors.*

Proof: Plug $h = \frac{\log n}{(c+1)\log\log n}$ into the bound from Lemma 3.4. Then Algorithm 6 runs in time $O(\log\log n)$ with $n \cdot 2^{\frac{\log n}{(c+1)\log\log n}} \cdot \log n$ processors. Since for sufficiently large n, $n \cdot 2^{\log n/((c+1)\log\log n)} \cdot \log n = n^{1+1/((c+1)\log\log n)} \cdot \log n \leq n^{1+1/c\log\log n}$, the bound follows. □

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 and $O(n \log^2 n)$ nodes. By Corollary 3.5, generating $\mathcal{N}$ takes $O(c \log\log n)$ time and uses $n^{1+1/c\log\log n}$ processors for arbitrary positive constant c. Performing pointer jumping on $\mathcal{N}$ takes $O(\log\log^2 n) = O(\log\log n)$ steps and uses $n \log^2 n$ processors. Thereby the algorithm designed fulfills the properties stated in Theorem 2.

4 Conclusions

As we already mentioned, our CREW PRAM algorithm uses hypergeometric random 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.

Our EREW algorithm can be significantly accelerated, if we remove the assumption that each permutation has to be generated with the same probability. Simply, in one parallel step one can set randomly the switches of the Beneš network and then determine the permutation defined by the network through pointer jumping in $O(\log \log n)$ steps. However the resulting probability distribution is far from being uniform. A challenging problem is to establish a lower bound for running time of uniform permutation generation on the strong randomized EREW PRAM.

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] 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.

[8] D. E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*, volume 2, Addison-Wesley, Reading, Massachusetts, 2nd edition, 1981.

[9] 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.

[10] 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.

[11] G. L. Miller and J. H. Reif, Parallel tree contraction, Part 1: Fundamentals, *Adv. in Comput. Res.* **5** (1989), 47–72.

[12] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.

[13] K. Mulmuley, *Computational Geometry. An Introduction Through Randomized Algorithms*, Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[14] J. Pieprzyk and B. Sadeghiyan, *Design of Hashing Algorithms*, Springer Verlag, Berlin, 1987.

[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.