Efficient Sampling of Random Permutations

Jens Gustedt

INRIA Lorraine & LORIA, France

Abstract

We show how to uniformly distribute data at random (not to be confounded with permutation routing) in two settings that are able to deal with massive data: *coarse grained parallelism* and *external memory*. In contrast to previously known work for parallel setups, our method is able to fulfill the three criteria of uniformity, work-optimality and balance among the processors simultaneously. To guarantee the uniformity we investigate the matrix of communication requests between the processors. We show that its distribution is a generalization of the multivariate hypergeometric distribution and we give algorithms to sample it efficiently in the two settings.

Key words: random permutations, random shuffling, coarse grained parallelism, external memory algorithms, uniformly generated communication matrix

1 Introduction and Overview

Random permutation [1] of data is a basic step in many computations. It is used e.g

- to achieve a distribution of the data to avoid load imbalances in parallel and distributed computing
- good generation of random samples to test algorithms and their implementations
- in statistical tests
- in computer games

to only name a few. Creating such permutations is relatively costly: to permute a vector of `long int`'s, we observed an average cost per item of about 60 to

URL: `http://www.loria.fr/ gustedt/` (Jens Gustedt).

[1] Sometimes also call random *shuffling*, see Knuth (1981).

100 clock cycles on commonly used architectures such as a 300 MHz Sparc or an 800 MHz Pentium III. One issue that causes this relatively high cost is the generation of (pseudo-)random numbers, but it is not the only one.

Procedure `Shuffle`(A, n) Sequential random permutation.

Input: A table A of n items.
Task: Permute the values in A at random
foreach $i = 0, \ldots, n-1$ **do**
    Choose a random integer j with $i < j < n$ uniformly at random
    Swap the items $A[i]$ and $A[j]$

`Shuffle` gives the classical algorithm to generate random permutations, see Moses and Oakford (1963); Durstenfeld (1964) and also (Knuth, 1981, Sec. 3.4.2). It obviously has a linear complexity, but it has the disadvantage that it addresses memory in an unpredictable way and thus causes a lot of cache misses. It is easy to see that the expected number of misses is close to the total number of items. So even in a standard memory bound setting, the running time of `Shuffle` is more or less proportional to the CPU-memory latency. In the above tests this bottleneck amounts to about 33% (Sparc) and 80% (Pentium) of the wall clock time. Applying this same algorithm to an external memory context that is able to treat massive amounts of data in large files would be completely infeasible.

Reducing the cost of such a time consuming subroutine is thus an issue, and here we will present an approach to achieve this. First of all, our approach is a parallel one, i.e uses several processors to sample a random permutation in a parallel or distributed setting. Then afterwards we will extend it to the sequential, IO-sensitive, framework.

When designing an alternative to the straightforward random generation of permutations, i.e to `Shuffle`, we have to ensure that we do not loose upon its quality: assuming that we have a "real" generator of random numbers we want each possible permutation to occur equally likely.

Our goal is to describe a realistic framework for the generation of random permutations. As a real suitable family of models of parallel computation we use a coarse grained setting that was derived from BSP, see Valiant (1990), which is called PRO, see Gebremedhin et al. (2002). PRO allows the design and analysis of scalable and resource-optimal algorithms. It is based on

- relative optimality compared to a fixed sequential algorithm as an integral part of the model;
- measuring the quality of the parallel algorithm according to its granularity, that is the relation between p and n.

In particular, coarseness here means $p \leq \sqrt{n}$, for p the number of processors and n the number of items. Later we will see how this $\sqrt{}$-restriction translates for the external IO setting.

PRO only considers algorithms that are both asymptotically work- and space-optimal when compared to the sequential one. This restrictive choice among possible parallel algorithms is made to ensure that the parallel algorithm is potentially useful. For real life implementations that are not work-optimal it is generally not even possible to have a speed up compared to a sequential program. So the use of such an algorithm (and an investment in a parallel machine) would not make much sense. As a consequence of this enforced optimality, a PRO-algorithm always yields linear speed-up relative to a reference sequential algorithm.

When respecting these prerequisites, synchronization cost between supersteps and communication latency between the processors such as they are measured in the BSP model are not an issue; these costs are dominated by bandwidth requirements, see Essaïdi and Gustedt (2006). So in addition, PRO only measures the coarse grained communication cost in terms of the bandwidth of the considered point-to-point interconnection network.

The use of such models is in contrast to the assumptions that are made by Czumaj et al. (1998) (which solve the problem by simulating some fine grained sorting network) and algorithms developed in the PRAM setting (see e.g Reif (1985); Hagerup (1991)). Since there the underlying models are fine grained and thus require a close synchronization between the processors for the treatment of each individual data item, they are not well suited for today's coarse grained architectures. Please also note, that the so-called *permutation routing* problem (see e.g Kruskal et al. (1990)) as it was intensively studied for the BSP and similar models is very different from our problem here. There one tries to optimize the communication of the messages during one superstep, the so-called h-relation.

Goodrich (1997) proposed an algorithm for our problem on the BSP architecture. Its main disadvantage is that it needs sorting of keys in the range of $[0 \dots n^2]$ to achieve its goal. If this is done with comparison based sorting it requires an overhead per item of $\log n$ and is thus not work optimal. When using involved radix sorting techniques instead, potential load imbalance between the processors becomes a problem.

Guérin Lassous and Thierry (2000) investigated several other algorithms to sample random permutations in a coarse grained setting. They found none that simultaneously fulfills the following criteria:

uniformity: Provided we have a perfect generator of randomness, all permutations must appear equally likely.

work-optimality: To be suitable for useful implementations, the total work (including communication and generation of random numbers) must at least asymptotically be the same as in a sequential setting.

balance: During the course of the algorithm none of the processors must be overloaded with work or data.

Especially uniformity and balance seem to work against each other. A typical trick to obtain balance for an algorithm that has some (small) probability of imbalance is to start-over whenever such an imbalance is detected. Usually this works well on the work-optimality when probabilities are small enough, and only increases the average running time a bit. But this also means that certain runs that lead to valid permutations are rejected. For such a procedure one then would have to guarantee not only that all permutations can be obtained, but that they all would be drawn with equal probability.

Another trick to avoid imbalance and non-uniformity is to iterate. If we have a method that is non-uniform but balanced we can iterate it to obtain a uniform distribution. Usually this needs a logarithmic number of iterations and so the total work is a log-factor away from optimality.

The goal of this paper is to close this gap by proving the following theorem:

Theorem 1 *There is a PRO-algorithm for computing a uniform random permutation that has an optimal grain compared to* `Shuffle`*: a network of p homogeneous processors may uniformly sample a random permutation of size $n = p \cdot m$, $p \leq m$. The usage of the following resources is $O(m)$ per processor and thus $O(n)$ in total: memory, bandwidth, computation time, and random numbers.*

Here we account for four different resources that to our opinion are the most significant in this setting: the use of random access *memory*, interprocessor communication *bandwidth*, general *computation time* and draws of *random numbers*. In particular, we think that it is important to distinguish the later two, since the quality of the solution to our problem inherently depends on the quality of the random generator that is used. Whereas pseudo-random generators could be accounted for by a constant number of operations, real devices of randomness (such as linux' `/dev/urandom`) tend to be much slower. So it is important to account for their cost separately.

The tools that we develop also allow us to go beyond that $\sqrt{}$-bound and formulate an IO-optimal algorithm for the problem.

Theorem 2 *There is an IO-efficient algorithm for computing a uniform random permutation: a machine with IO-block size B and an internal memory of m IO-blocks may uniformly sample a random permutation of $N = nB$ items with $O(n \log_m n)$ IO-operations.*

The organization of this paper is as follows. Section 2 introduces the idea of separating the sampling of a $p \times p$ communication matrix between the processors and the generation of the permutation itself. Section 3 discusses the probability distribution of such communication matrices. Sections 4 and 5 then provide sequential and parallel algorithms to sample such a matrix. Section 6 gives the extension to the external IO setting and Section 7 concludes by briefly discussing some experiments and tests.

2 Simulating random permutations by the number of elements distributed between the processors

Given a vector v of size n our goal is to sample a random permutation v' of v. We assume that v is distributed with m_i elements on processor P_i, i.e that

$$n = \sum_{i=1,\ldots,p} m_i, \tag{1}$$

and that the newly permuted vector v' should be distributed alike. To be able to better describe the distribution and to give algorithms to simulate it, it will be convenient to generalize the problem: we assume that we have p source-chunks[2] C_i that send the data to p' target-chunks C'_j such that the chunk sizes of the source array (i.e the array before performing the permutation) are $m_1, \ldots, m_p$ and of the target array are $m'_1, \ldots, m'_{p'}$.

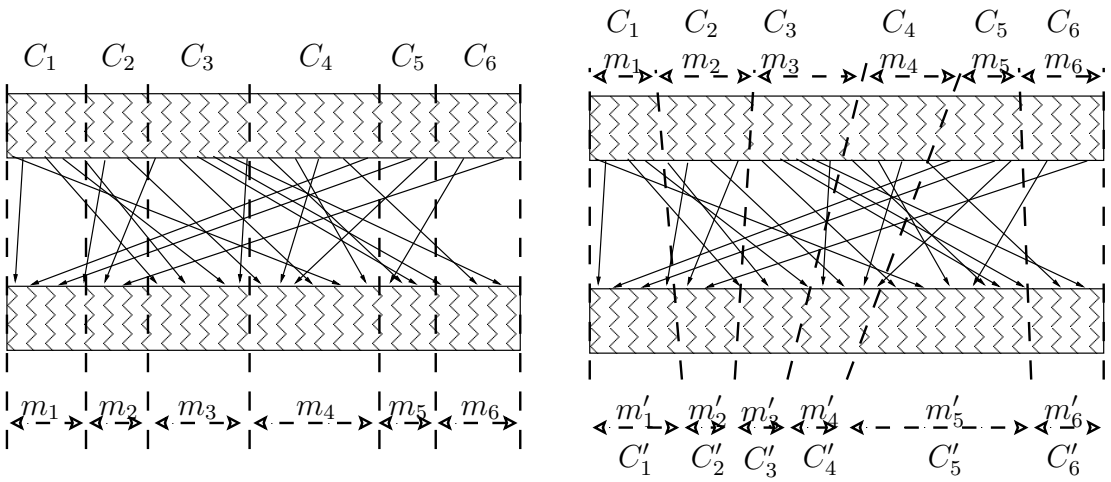
Problem 1 (Random Permutation with chunks)

Input: *Vector v of n items in total that is distributed on p source processors P_i, $i = 1, \ldots, p$, such that processor P_i holds a chunk C_i of m_i elements; target vector v' of the same length n such that target processor P'_j, $j = 1, \ldots, p'$ holds a chunk B'_j of m'_j items.*

Task: *Redistribute the elements of v into v' such that every permutation is equally likely.*

We achieve our goal by first computing a matrix $A = (a_{ij})$ that accounts for the amount of communication from chunk C_i to chunk C'_j. A is not an arbitrary matrix but has special properties. We have

[2] We refer to *chunks* as large contiguous parts of the data. This is to distinguish them from *blocks* which will be used to describe minimal units of communication in the external IO setting.



(a) with identical chunk sizes

(b) with different chunk sizes and groups of processors

Fig. 1. Permutation from 6 source chunks to 6 target chunks

$$m_i = \sum_{j=1,p'} a_{i,j}, \text{ for all } i = 1, \ldots, p \tag{2}$$

$$m'_j = \sum_{i=1,p} a_{i,j}, \text{ for all } j = 1, \ldots, p'. \tag{3}$$

A permutation can then be realized by using the matrix A to send out a_{ij} items between all pairs of processors. As we aim for a random permutation we must ensure that

- the individual items that are sent from P_i to P'_j are chosen arbitrarily and that
- all items that are received on P'_j are mixed randomly.

This can easily be achieved by two local random permutations, one before the communication and the other thereafter, see Procedure `ParPerm`.

Obviously, all matrices with properties (2) and (3) may arise as such a communication matrix: it is easy to set up a permutation that exactly achieves such a matrix. So we easily get the following proposition.

Proposition 1 *Procedure* `ParPerm` *is correct. Besides the computation of the matrix A the algorithm is balanced and asymptotically work-optimal.*

Proof: A particular matrix might look unbalanced and send quite different amounts of items between different pairs of processors. Equations (2) and (3) guarantee that the amount that each processor sends and receives stays under control. So if the send and receive operations are done without blocking, the communication phase stays balanced. □

Procedure `ParPerm`$(C_i, (m_i), C'_i, (m'_i))$: Parallel Random Permutation

Input: Each source processor P_i for $i = 1, \ldots, p$ has a chunk C_i of m_i input elements and each target processor P'_j has a chunk C'_i of length m'_i to hold the result.

Output: The elements in all C_i are globally permuted into the vectors C'_i such that any permutation appears equally likely.

foreach P_i, $i = 1, \ldots, p$ **do** permute C_i locally in parallel
Choose $A = (a_{i,j})$ according to (2) and (3)
foreach P_i, $i = 1, \ldots, p$ **do**
  for $j = 1, \ldots, p'$ **do** send $a_{i,j}$ items to processor P'_j
foreach P'_j, $j = 1, \ldots, p'$ **do**
  for $i = 1, \ldots, p$ **do** receive $a_{i,j}$ items from processor P_i
foreach P'_j, $j = 1, \ldots, p'$ **do** permute C'_i locally in parallel

If we want the distribution of the resulting permutations to be uniform not all possible matrices A occur with the same probability. This can already be observed for the permutation of 4 items 12|34 that we separate into to two source and target chunks of size two, each:

12|34	12|43	13|24	13|42	14|23	14|32	21|34	21|43	23|14	23|41	24|13	24|31
2 0 0 2	2 0 0 2	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	2 0 0 2	2 0 0 2	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1
31|24	**31|42**	**32|14**	**32|41**	**34|12**	**34|21**	**41|23**	**41|32**	**42|13**	**42|31**	**43|12**	**43|21**
1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	0 2 2 0	0 2 2 0	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	0 2 2 0	0 2 2 0

We observe that of the three possible matrices, two occur only four times each whereas the third occurs 16 times. For such a random permutation of 4 items it is unlikely (probability 1/6) that the items of each chunk stay in the same. And for the general problem the (unique) diagonal matrix that describes the event that all chunks keep their items occurs with a very small probability, only.

The distribution of the matrix itself under these assumptions is not trivial and in particular the different a_{ij} are not independent. In the example, each of the occurring matrices can in fact be determined from the upper left entry, say. In an important part of this paper we will investigate how to get our hand on these matrices and how to generate them randomly with the desired distribution. The generation has to be done in such a way that it does not dominate the running time of Procedure `ParPerm`.

Problem 2 (Random Communication Matrix)

Input: *Vectors m_i and m'_i as for Problem 1.*

Output: *A random choice of communication matrix A with properties (2) and (3) such that all matrices occur with the probability which is induced by Prob-*

lem 1, i.e *by first choosing a random permutation and computing its communication matrix a posteriori.*

Proposition 2 *If the communication matrix A in Procedure* `ParPerm` *is chosen according to Problem 2 it provides a solution to Problem 1.*

Proof: If we fix some communication matrix A, then the local permutations before and after the communication ensure that all permutations that realize A are generated with the same probability. Since A in turn is supposed to be chosen with the right probability the correctness follows. □

In Sections 4 and 5 we will then give algorithms that will prove the following theorem, and thus conclude the proof of Theorem 1.

Theorem 3

(1) There is a sequential algorithm that samples a matrix as described in Problem 2 with linear cost, i.e $O(p^2)$.
(2) A network of p homogeneous processors may sample such a matrix such that the usage of the following resources is $O(p)$ per processor and thus $O(p^2)$ in total: memory, computation time, random numbers and bandwidth.

3 The probability distribution of A

We will see that the distribution of communication matrix A is closely related to the so-called hypergeometric distribution $h(t, w, b)$. This is the distribution of an urn experiment $X_{t,w,b}$ where we draw t balls out of w "*white*" and b "*black*" balls and measure the outcome of the number of *whites* [3]. It has the probability

$$P\left(X_{t,w,b} = k\right) = \frac{\binom{w}{k}\binom{b}{t-k}}{\binom{w+b}{t}} \tag{4}$$

The hypergeometric distribution can be sampled quite efficiently, see Stadlober and Zechner (1999); Zechner (1997); Zechner and Stadlober (1993). Its computational cost is dominated by the calls to a random generator subroutine. For the experiments that are described in Essaïdi and Gustedt (2006) the amount of calls to the random generator (POSIX' `erand48` in our case) that were issued by one call of h was less than 1.5 on average and strictly below 10 for the worst case.

[3] The reader that is not familiar with these notions might refer to Siegrist (2001).

Proposition 3 *Each individual entry $a_{i,j}$ of communication matrix $A = (a_{i,j})$ obeys a hypergeometric distribution $h(m'_j, m_i, n-m_i)$ where $n = \sum m_i = \sum m'_i$.*

Proof: The element $a_{i,j}$ reflects the number of elements that processor P_i sends to processor P_j. We consider the elements on C_i as being "white", $w = m_i$, and all the others as being "black", $b = n - w = n - m_i$ and set $t = m'_j$. A random permutation π chooses to send all $\binom{n}{t}$ subsets of cardinality t with equal probability to C'_i. Among those t-subsets there are $\binom{w}{k}\binom{b}{t-k}$ that have exactly k white elements. □

The special case when the matrix is actually a row (or column) is also known as the *multivariate hypergeometric distribution*, see e.g. Siegrist (2001) for the terms.

The matrix A has another interesting property, namely that it is self-similar to sums of submatrices.

Proposition 4 *Let A be as above, $0 = i_0 < i_1 < \cdots < i_q = p$ and $0 = j_0 < j_1 < \cdots < j_{q'} = p'$. Then the matrix*

$$\mathfrak{A} = (\mathfrak{a}_{r,s})_{\substack{r=1,\ldots,q \\ s=1,\ldots,q'}} \quad \textit{for} \quad \mathfrak{a}_{r,s} = \sum_{\substack{i_{r-1}<i\le i_r \\ j_{s-1}<j\le j_s}} a_{i,j}$$

is distributed as for the problem with input chunks of size

$$\mathfrak{m}_r = \sum_{i_{r-1}<i\le i_r} m_i \quad \textit{and} \quad \mathfrak{m}'_s = \sum_{j_{s-1}<j\le j_s} m'_j.$$

Proof: This follows directly by the fact that we may join the input chunks C_i and output chunks C'_j according to the super-indices i_r and j_s. □

In view of Proposition 4, Proposition 3 has an easy generalization.

Proposition 5 *Let $0 = i_0 < i_1 < \cdots < i_q \le p$ and $0 = j_0 < j_1 < \cdots < j_{q'} \le p'$ and $\mathfrak{A} = (\mathfrak{a}_{r,s})$ be as above. Then each $\mathfrak{a}_{r,s}$ is distributed with $h(t, w, b)$ with*

$$t = \mathfrak{m}'_{r,s} \tag{5}$$
$$w = \mathfrak{m}_{r,s} \tag{6}$$
$$b = n - \mathfrak{m}_{r,s} \tag{7}$$

Having identified the individual distribution for each entry of A is not sufficient to efficiently draw samples for A. This can already be seen if we only consider $p = 2$ as in the example of Section 2. Once we have chosen $a_{1,1} = k$, all the

three other matrix entries are already determined by (2) and (3). Namely we have

$$A|_{a_{1,1}=k} = \left(\begin{array}{c|c} k & m_1 - k \\ \hline m'_1 - k & n - (m'_1 + m_1) + k \end{array}\right). \tag{8}$$

So for this special case the rest of the matrix is already completely determined by the first value and in general the same holds for the last column and row of the matrix which are determined by the other columns and rows respectively.

In general we split our matrix at some index i_1 into an upper and lower part and describe the relative influence of the outcome in both parts by some suitable conditional probability. All algorithms that we will present will be based on this principle: first they will sample some (possibly multivariate) hypergeometric distribution to describe the split of the problem and then they will proceed to solve the two parts independently.

Proposition 6 *Let A be as above, $0 = i_0 < i_1 < i_2 = p$, $j_s = s$ for $s = 1, \ldots, p'$. Suppose $(\mathfrak{a}_{1,s})$ as defined above has the outcome (α_s) then the conditional distribution for the upper half of matrix A is the same as for the problem with input $m_1, \ldots, m_{i_1}$ and $\alpha_1, \ldots, \alpha_{p'}$.*

Clearly an analogous claim holds for the outcome of the lower half and for a split of the matrix into a left and right half.

4 Sequential algorithms to sample A

From Proposition 6 with choice of $i_1 = p - 1$ we get a first sequential algorithm that samples a communication matrix. We need the special case (Procedure `MultVarHyp`) where the matrix consists of exactly one row (or column) (the *multivariate hypergeometric distribution*) as a subroutine for the general one. A first algorithm to solve the general problem is then summarized as Procedure `SampMat`.

For the multivariate hypergeometric distribution we want to place $\mathfrak{m}$ items into $\mathfrak{b}$ bins. The $\mathfrak{b}$ bins are grouped together into p chunks that hold $b_1, \ldots, b_p$ bins. The goal is to know how many items will fall into each of the chunks. The idea of `MultVarHyp` is to choose the number toRight among the $\mathfrak{m}$ items that will go to the part "on the right", *i.e.* with indices greater than i. Observe that $\alpha_i = \mathfrak{m} - \texttt{toRight}$ is then distributed according to $h(b_i, \mathfrak{b} - \mathfrak{m}, \mathfrak{b})$.

`SampMat` generalizes the main idea to sample a whole matrix. The matrix is sampled row by row, starting with the last row. Instead of one value "toUp" we now have to compute a vector of the number of items for each column.

Procedure `MultVarHyp`$(p, \mathfrak{b}, \mathfrak{m}, (b_i))$: Sequential sampling of a multivariate hypergeometric distribution

Input: Integers p, $\mathfrak{b}$ and $\mathfrak{m}$, a vector $(b_1, \ldots, b_p)$ with $\mathfrak{m} \leq \mathfrak{b} = \sum_{i=1,\ldots,p} b_i$.
Output: Random vector (α_i) distributed with a multivariate hypergeometric distribution with parameters $\mathfrak{m}$ and (b_i).

for $i = 1, \ldots, p$ **do**
  Choose `toRight` according to $h(\mathfrak{b} - b_i, \mathfrak{m}, \mathfrak{b} - \mathfrak{m})$
  Set $\alpha_i = \mathfrak{m} - \texttt{toRight}$, $\mathfrak{m} = \texttt{toRight}$ and $\mathfrak{b} = \mathfrak{b} - b_i$.

Procedure `SampMat`$(p, (m_i), p', (m'_i))$: Sequential sampling of a communication matrix

Input: Integer p, vector of p values (m_i), integer p' and vector of p' values (m'_i).
Output: Random communication matrix $(a_{i,j})$ such that all such matrices appear with the probability corresponding to the number of permutations that realize them.

for $i = p, \ldots, 1$ **do**
split Choose vector $(\texttt{toUp}_1, \ldots, \texttt{toUp}_{p'})$ according to a multivariate hypergeometric distribution with parameters m_i and $(m'_1, \ldots, m'_{p'})$
update **for** $j = 1, \ldots, p'$ **do**
    $a_{i,j} = m'_j - \texttt{toUp}_j$
    $m'_j = \texttt{toUp}_j$

Proposition 7 `SampMat` *samples* $(a_{i,j})$ *with* $O(p \cdot p')$ *basic operations and* $O(p \cdot p')$ *samples of the hypergeometric distribution* h.

Proof: For the correctness observe that the algorithm is a direct application of Proposition 6.

For the complexity it is clear that each call to `MultVarHyp` in **split** costs $O(p')$. Since **update** also goes in $O(p')$ and the loop is done p times this proves the claim. □

To go a step towards a possible parallelization we give a recursive algorithm for the same task, see Procedure `RecMat`. Observe that `MultVarHyp` could be seen as an iterative variant of `RecMat` for the special choice of $q = 1$. The recursive formulation also has the advantage that we may split the input for the samples of the hypergeometric distribution more or less evenly.

Procedure `RecMat`$(p, (m_i), p', (m'_j))$: Recursive sampling of a communication matrix

Input: Integer p, vector of p values (m_i), integer p' and vector of p' values (m'_i).

Output: Random communication matrix $(a_{i,j})$ such that all such matrices appear with the probability corresponding to the number of permutations that realize them.

if $p < 2$ **then return** $(m'_j)_{j=1,\dots,p'}$ **else**

- Choose an index $0 < q < p$ and set $t = \sum_{q \le i < p} m_i$
- Choose vector $(\texttt{toUp}_j)$ according to a multivariate hypergeometric distribution with parameters t and (m'_j)
- Set vector $(\texttt{toLo}_j)$ to $(m'_j - \texttt{toUp}_j)$
- Sample $(a_{i,.})_{i=1,\dots,q-1}$ with `RecMat`$(q, (m_i), p', (\texttt{toLo}))$
- Sample $(a_{i,.})_{i=q,\dots,p}$ with `RecMat`$(p-q, (m_{i+q}), p', (\texttt{toUp}))$

5 Parallel algorithms

We will derive our first parallel algorithm from `RecMat` by taking care of the fact that we cannot assume that one of the processors may hold the whole communication matrix. For the parallel algorithm we will focus on the symmetric case where $p = p'$ is the number of processors and such that all processors have the same local share $M = n/p$ of the vector that will be permuted. The reader may easily adapt this algorithm to the general situation.

Proposition 8 `ParaMatLog` *is correct. The running time, communication cost per processor and number of samples of the hypergeometric distribution h per processor are* $\Theta(p \log p)$*. The total work load, total communication and total number of samples of h are* $\Theta(p^2 \log p)$ *each.*

Proof: For the correctness observe that the **while**-loop iteratively divides the processor range in halves $r \le \texttt{id} < s$. The main work of what would be a recursive call in `RecMat` is always done by the "head" processor P_r. In each iteration, P_r updates its local data and the new head P_q of the upper half of the range receives the necessary data. So they are both able to perform the computation in the next iteration correctly.

For the complexity observe that the **while**-loop is executed $O(\log p)$ times. □

Observe that this does not give us a work-optimal algorithm for sampling of the matrix. We have a $\log p$-factor both in the time and in the cost. But for solving the permutation problem this matrix generation can already be useful. The communication cost for the exchange of the data dominates the running time as long as $p \log p < M = n/p \iff p^2 \log p < n$. So with `ParaMatLog` we

Procedure `ParaMatLog`$(p, \mathtt{id}, M)$: Parallel sampling of a communication matrix (with a log-factor in the total work)

Input: Total number of processors p, processor `id` with $1 \leq \mathtt{id} \leq p$ and value M.

Result: For each processor P_i a vector (β_i) representing a row of the random communication matrix $(a_{i,j})$ such that all such matrices appear with the probability corresponding to the number of permutations that realize them.

```
if id = 1 then Initialize (β_i) ≡ M
r = 1 and s = p + 1
while s − r > 1 do
    q = ⌊(r + s)/2⌋
    switch id do
        case r:
            t = (s − q)M
            Choose vector (toUp_i) according to a multivariate hypergeometric
            distribution with parameters t and (β_i)
            Send (toUp_i) to processor P_q
            (β_i) = (β_i − toUp_i)
        case q: Receive (β_i) from processor P_r
        otherwise do nothing
    if id ≤ q then r = q else s = q
```

are only a log-factor away from the optimal granularity of $\sqrt{n}$.

`ParaMat` avoids this extra log-factor. The difference between the two algorithms is that here the matrix is not only sliced in one dimension. In alternation, they are split along both dimensions. This is controlled by variables Δ and ∇ that are set to symbolic values $\boxminus$ and $\boxplus$. Also the index computation gets more involved, r, s and q now control the processor indices, whereas $r^{\boxplus}$, $s^{\boxplus}$ and $q^{\boxplus}$ control the indices for the vertical split and $r^{\boxminus}$, $s^{\boxminus}$ and $q^{\boxminus}$ control those for the horizontal split.

Proposition 9 `ParaMat` *is correct. The running time, communication and samples of the hypergeometric distribution h are $\Theta(p)$ per processor and $\Theta(p^2)$ in total.*

Proof: The proof is analogous to that of Proposition 8. At the end of the **while**-loop every processor is left with a sub-matrix that it has to sample. The test of the **while**-loop guarantees that the size of this matrix is $O(p)$. This shows in particular that all processors only have to handle data of the size of $O(p)$.

For the complexity observe that the **while**-loop is still executed $O(\log p)$ times but that the time for each iteration diminishes. In fact, the total size of both

Procedure `ParaMat`(p, id, M): Cost-optimal parallel sampling of a communication matrix

Input: Total number of processors p, processor `id` with $1 \leq \texttt{id} \leq p$ and value M.

Result: A vector representing a row of random communication matrix $(a_{i,j})$ such that all such matrices appear with the probability corresponding to the number of permutations that realize them.

1 **if** $id = 1$ **then** Initialize $(\beta_i^{\square\square})$ and $(\beta_i^{\square\!\square})$ to M
$\Delta = \square\square$ and $\nabla = \square\!\square$
Set $r, r^{\square\square}, r^{\square\!\square}$ to 1 and $s, s^{\square\square}, s^{\square\!\square}$ to $p+1$

2 **while** $s^{\square\square} - r^{\square\square} > \sqrt{p}$ *or* $s^{\square\!\square} - r^{\square\!\square} > \sqrt{p}$ **do**
- $q = \lfloor (r+s)/2 \rfloor$
- $q_\Delta = \lfloor (r_\Delta + s_\Delta)/2 \rfloor$
- **switch** id **do**
  - **case** *r:*
    - $t = \sum_{q_\Delta \leq i < s_\Delta} \beta_i^\Delta$
    - **foreach** $q_\Delta \leq i < s_\Delta$ **do** Send β_i^Δ to processor P_q
    - **for** *the range* $r_\nabla \leq i < s_\nabla$ **do**
      - Choose vector $(\texttt{toDelta}_i)$ according to a multivariate hypergeometric distribution with parameters t and (β_i^∇)
    - **foreach** $r_\nabla \leq i < s_\nabla$ **do**
      - Send $\texttt{toDelta}_i$ to processor P_q
      - $\beta_i^\nabla = \beta_i^\nabla - \texttt{toDelta}_i$
    - **foreach** $q_\Delta \leq i < s_\Delta$ **do** Send β_i^Δ to processor P_q
  - **case** *q:*
    - **foreach** $q_\Delta \leq i < s_\Delta$ **do** Receive β_i^Δ from P_r
    - **foreach** $r_\nabla \leq i < s_\nabla$ **do** Receive β_i^∇ from P_r
  - **otherwise** *do nothing*
- **if** $id < q$ **then**
  - $r = q$
  - $r_\Delta = q_\Delta$
- **else**
  - $s = q$
  - $s_\Delta = q_\Delta$
- Swap the values of Δ and ∇

3 Sequentially sample the submatrix $(a_{i,j})$ of the communication matrix with indices $r^{\square\square} \leq i < s^{\square\square}$ and $r^{\square\!\square} \leq j < s^{\square\!\square}$ according to the input vectors $(\beta_i^{\square\square})_{r^{\square\square} \leq i < s^{\square\square}}$ and $(\beta_i^{\square\!\square})_{r^{\square\!\square} \leq i < s^{\square\!\square}}$.

4 Redistribute the matrix such that each processor holds the row $(a_{\texttt{id},\cdot})$

ranges halves with every second iteration. So by a standard halving argument the total time for the **while**-loops is $O(p)$.

Sampling the submatrix (by Section 4) and the final data communication is linear in p, so this concludes the proof. □

Proof of Theorem 1: It is straightforward to see that `ParPerm` when applied to the case that $m_0 = \cdots = m_{p-1} = m'_0 = \cdots = m'_{p-1} = \sqrt{n}$ has linear cost on each processor for sampling the two local permutations and for sending and receiving messages.

Theorem 3 now provides us with the necessary tool to sample the communication matrix. `ParaMat` samples matrices with the needed probability distribution (giving correctness) and has a cost of $O(p)$ per processor. Thus as long as $p \leq n$, `ParaMat` does not change the complexity of `ParPerm`, which stays linear in n. So, in particular `ParPerm` is a PRO algorithm with an optimal grain compared to the sequential reference algorithm `Shuffle`. □

6 IO efficiency

In view of the idea to use efficient coarse grained algorithms also for the context of external memory, see Cormen and Goodrich (1996); Dehne et al. (1997), our ideas are not only useful for parallel algorithms but can also be used in a setting which is concerned about IO, see also Gustedt (2003). In this section we will show how our algorithms performs in view of IO considerations.

The now classical IO model, see Vitter (2001), captures the constraints that modern architectures impose when handling massive data. Its main characteristics are that it supposes that accessible memory is divided into a small and nearby part of size M, called internal memory, that is directly accessible and a large and distant part of size N, called external memory. Usually this model is imagined as handling RAM as internal memory and disk space as external memory, but may also represent any two consecutive levels of the memory hierarchy.

IO to the external memory is done block by block, with blocks of size B. The complexity of an algorithm is then expressed in the amount of block-IO of the algorithm. [4]

The average-case and worst-case number of I/Os required for permuting N

[4] Generally the model also includes the case that the external memory is split in D independent *disks*. Here we will assume that $D = 1$.

data items, see Vitter and Aggarwal (1988), is essentially the same as for sorting, namely

$$\Theta\left(\min\left\{N, n \cdot \log_m n\right\}\right), \tag{9}$$

where $m = \frac{M}{B}$ and $n = \frac{N}{B}$ are the number of blocks of the internal memory and the data, respectively.

A parallel coarse grained algorithm in PRO immediately translates into an algorithm for external memory: we just have to divide the data into chunks to which we attribute a 'virtual' processor. We then simulate the different processors in a round robin fashion, one superstep after the other. Interprocessor communication then translates into IO on the external medium. It has been shown in Gustedt (2003) that the constraints of PRO guarantee that the IO in such a setting behaves nicely: the write operations accumulate large enough chunks of data such that the number of block IOs is proportional to the overall communication of the PRO algorithm accounted in blocks.

At a first glance this seems to show that we need less block IO, namely linear, than the bound in (9). But as we have that p is at most $\sqrt{N}$ we also have that M has to be at least $\sqrt{N}$. So in fact

$$1 < \log_m n = \frac{\log N - \log B}{\log M - \log B} < \frac{\log N}{\log M} < \frac{\log N}{\log \sqrt{N}} = 2. \tag{10}$$

Observe that this bound of 2 is independent of the block size B.

A straightforward transcription of `ParPerm` is given in Procedure `IOPermEq`.

Procedure `IOPermEq`(F, F', N): Sequential Random Permutation with equal chunk sizes

---

Input: File-pointer F of N items.
Output: A random permutation of the items written in file-pointer F'
Split the input and output file into $p = \sqrt{N}$ chunks C_i respectively C'_i of approximately equal size
Choose $A = (a_{i,j})$ according to (2) and (3)
foreach $i = 1, \ldots, p$ **do**
  Load C_i into internal memory
  Permute C_i
  for $j = 1, \ldots, p$ **do**
    Write $a_{i,j}$ items from C_i to C'_j.
foreach $j = 1, \ldots, p$ **do**
  Load C'_j into internal memory
  Permute C'_j
  Write C'_j back to the external memory

Proposition 10 *If for the sizes N and M of the external and internal memory $\sqrt{N} \leq M \leq N/2$, Procedure* `IOPermEq` *performs with $\Theta(N/B)$ IO operations.*

Proof: It is easy to see that the choice of the matrix A can be done efficiently with $n = N/B$ IO operations, so the claim follows directly. □

This algorithm is not yet suitable for a generalization where the respective sizes N and M are arbitrary. The main problem here is that the data is permuted twice in internal memory, so if we would simply apply recursion we would increase the complexity. Another issue is then the computation of the matrix A. To avoid unnecessary IO operations it will be better to sample it incrementally, line by line, as we will use it.

`IOPerm` is such a procedure. It uses different sizes for the chunks on the 'input' and 'output' side. The idea is to choose the chunks on the input side small, such that each chunk fit into internal memory at once. Then we may permute these input chunks directly without going into recursion. On the output side, the chunks are large but few, so few that at any time we may hold one IO-block from each output chunk in internal memory.

The following propostion states that in fact `IOPerm` is an algorithm as claimed by Theorem 2, and thus the proof of the proposition concludes the proof of the theorem, too.

Proposition 11 *If m the number of IO-blocks in internal memory is at least 2, Procedure* `IOPerm` *performs with $O(n \log_m n)$ IO operations.*

Proof: For simplicity, we will first assume that $m, B \geq 4$ and briefly point out below how this restriction may be relaxed. First, we show that all operations of the procedure can be performed in internal memory. For the **if**-part this is direct, since here F is small enough to fit into internal memory.

For the **else**-part we first observe that for any of the output chunks C_i'' we need at most one IO-block in internal memory at a time: the write operations in **write** may write IO-blocks as they are filled completely to the file and only have to keep one fractional part per chunk between iterations. So the number of blocks needed for this are

$$p' = n/t' \leq \frac{nm}{2n} = m/2. \tag{11}$$

The two vectors $a_{i,\cdot}$ and `toLo` have a length of t' words, each. Thus each of them occupies $p' < m/2$ machine words and both together $m/B \leq m/4$ blocks. Finally, each of the C_i has less than $m/4$ blocks and so in total we use at most $m/2 + m/4 + m/4 = m$ blocks simultaneously.

Procedure `IOPerm`(F, F', N, w, B, m) IO-optimal Random Permutation

Input: File-pointers F and F' to N items, the number w of machine words per item, the IO-block size B in machine words, and the size of the internal memory m in number of IO-blocks.

Output: A random permutation of the items written to file-pointer F'.

Let $n = \lceil Nw/B \rceil$ be the number of IO-blocks in F

if $n \leq m$ **then**
- Load F into internal memory
- Permute F
- Write F to F'

else
- Create an auxiliary file F'' for N items.
- Let $t = \max\{2, \lfloor m/4 \rfloor\}$. Split the input file into p chunks C_i of t IO-blocks.
- Let $t' = \min\{\lceil n/2 \rceil, \lceil 2n/m \rceil\}$. Split F' and F'' into p' chunks C'_j and C''_i of t' IO-blocks.
- Initialize a vector `toLo` with the p' sizes of the C'_i.
- **foreach** $i = 1, \ldots, p$ **do**
  - Sample a row $a_{i,\cdot}$ of the communication matrix and update `toLo`.
  - Load C_i into internal memory.
  - Permute C_i.
  - **for** $j = 1, \ldots, p$ **do**
    - **write** Write $a_{i,j}$ items from C_i to C''_j.
- **foreach** $j = 1, \ldots, p'$ **do**
  - `IOPerm`$(C''_j, C'_j, wt', w, B, m)$
- Delete F''.

For the IO-complexity of the procedure itself, not accounting for recursion, note that we read each input block exactly once. Output blocks that are entirely part of a chunk are also only written once when they are full. There are at most $2p' < m/2 < n/2$ partial blocks at the beginning or end of the output chunks, so the IO is $O(n)$.

For the recursion, the choice of t' ensures that the chunks are always split on IO-block boundaries and so we don't have to consider partial blocks for the computation of the problems sizes that go into recursion. So the total amount of IO-blocks that are handled is n for each level of recursion. Since the recursion level is easily seen to be $O(\log_m n)$, the claim for the case that $m, B \geq 4$ follows.

For $B < 4$ we have to ensure that the parts of the matrix that we store at a given time are not too large. This can be done by reading `toLo` into internal memory and using MultVarHyp to sample $a_{i,\cdot}$ and update `toLo`. Then $a_{i,\cdot}$ and `toLo` can be written to external memory and the parts of $a_{i,\cdot}$ that are needed

afterwards can be read as we need it. All these reads and writes are at most $O(p') = O(m/4) = O(m)$ IO-operations.

In the case of $2 \leq m < 4$ we have that $t = 1$ and $p' = 2$. It is easy to see that then each of the $p = n$ iterations can be done with 3 read and 4 write operations. So the number of IOs is again $O(n)$ when not accounting for recursive calls. The recursion depth is $\log_2 n \leq log_2 m \cdot \log_m n \leq log_2 4 \cdot \log_m n = 2 \log_m n$, which proves the claim. □

7 Conclusion and Outlook

In the present paper we presented algorithms that for the first time allow for the efficient generation of randomized permutations that simultaneously fulfill the three criteria of uniformity, work-optimality and balance. Only the combination of these three properties guarantees the usefulness of such an algorithm in a practical setting:

- only a uniform distribution gives sufficient quality on the randomness of the permutations such that they allow for unbiased simulations and fair games
- only work-optimality and balance together guarantee a linear speed-up such that the running time of applications is not dominated by the generation of random input samples.

The main effort that we had to invest in this paper was to provide a subroutine needed in Procedure `ParPerm`: it uniformly samples the communication matrix, the matrix that describes the amount of items that each pair of processors has to exchange.

In fact, the algorithms as presented here translate to straightforward implementations. Part of the algorithms (sequential sampling of the matrix, only) were implemented and then tested on different platforms (Sun Sparc, Intel Pentium Linux, SGI IRIX) with up to 128 processors and 5 billion items. The framework for the implementation was SSCRAP (now integrated into parXXL), see Essaïdi et al. (2002); Gustedt et al. (2006), our environment for coarse grained algorithms. To our complete satisfaction, this generation of permutations is now a solid basis for randomized performance tests and benchmarks of the library.

The overhead due to the parallelization over the simple sequential algorithm is a factor between 2 and 3 as one would expect: we have to perform two local permutations and the communication between the processors. Therefore the absolute efficiency of a parallel implementation of algorithm will at most be between 30 and 50%.

The tests showed that the main limitation for Procedure `ParPerm` when run on large data sets is the communication phase, even when executed on a shared memory machine. On the other hand, for smaller data sets, the computation of the matrix might be a bottleneck. So in situations where medium sized permutations are needed repeatedly a parallel implementation of the matrix sampling will be helpful.

SSCRAP also allows for the efficient simulation of parallel algorithms in an external memory setting. Tests in such a setting with the algorithms as presented in this paper have been reported in Gustedt (2003).

Acknowledgment

The author thanks Kjell Kongsvik and Mohamed Essaïdi for their help with the implementation, and an anomynous referee for his very helpful suggestions to improve the writing of this paper.

References

Cormen, T. H., Goodrich, M. T., 1996. A bridging model for parallel computation, communication, and I/O. ACM Computing Surveys 28A (4).

Czumaj, A., Kanarek, P., Kutylowski, M., Lorys, K., 1998. Fast generation of random permutations via networks simulation. Algorithmica 21 (1), 2–20.

Dehne, F. K. H. A., Dittrich, W., Hutchinson, D., 1997. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. In: ACM Symposium on Parallel Algorithms and Architectures. pp. 106–115.

Durstenfeld, R., 1964. Algorithm 235: Random permutation. Commun. ACM , 420.

Essaïdi, M., Guérin Lassous, I., Gustedt, J., 2002. SSCRAP: An environment for coarse grained algorithms. In: Fourteenth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002). pp. 398–403.

Essaïdi, M., Gustedt, J., Feb. 2006. An experimental validation of the PRO model for parallel and distributed computation. In: Di Martino, B. (Ed.), 14th Euromicro Conference on Parallel, Distributed and Network based Processing. IEEE, The Institute of Electrical and Electronics Engineers, pp. 449–456.

Gebremedhin, A. H., Guérin Lassous, I., Gustedt, J., Telle, J. A., 2002. PRO: a model for parallel resource-optimal computation. In: 16th Annual International Symposium on High Performance Computing Systems and Applications. IEEE, The Institute of Electrical and Electronics Engineers, pp. 106–113.

Goodrich, M. T., 1997. Randomized fully-scalable BSP techniques for multi-searching and convex hull construction. In: Saks, M., et al. (Eds.), Proceedings of the eighth annual ACM-SIAM Symposium on Discrete Algorithms. SIAM, Society of Industrial and Applied Mathematics, pp. 767–776.

Guérin Lassous, I., Thierry, É., 2000. Generating random permutations in the framework of parallel coarse grained models. In: Proceedings of OPODIS'2000. Vol. 2 of Studia Informatica Universalis. pp. 1–16.

Gustedt, J., 2003. Towards realistic implementations of external memory algorithms using a coarse grained paradigm. In: International Conference on Computational Science and its Applications (ICCSA 2003), part II. No. 2668 in LNCS. Springer, pp. 269–278.

Gustedt, J., Vialle, S., De Vivo, A., 2006. parXXL: A fine grained development environment on coarse grained architectures. In: Kågström, B., Elmroth, E., Dongarra, J., Wasniewski, J. (Eds.), Workshop on state-of-the-art in scientific and parallel computing (PARA'06). *to appear*.
URL http://www.hpc2n.umu.se/para06/papers/paper_48.pdf

Hagerup, T., 1991. Fast parallel generation of random permutations. In: Automata, Languages and Programming. Vol. 510 of Lecture Notes in Comp. Sci. Springer-Verlag, pp. 405–416, proceedings of the 18th International Colloquium ICALP'91.

Knuth, D. E., 1981. The Art of Computer Programming, 2nd Edition. Vol. 2: Seminumerical Algorithms. Addison-Wesley.

Kruskal, C. P., Rudolph, L., Snir, M., Mar. 1990. A complexity theory of efficient parallel algorithms. Theoretical Computer Science 71 (1), 95–132.

Moses, L. E., Oakford, R. V., 1963. Tables of Random Permutations. Stanford University Press.

Reif, J. H., 1985. An optimal parallel algorithm for integer sorting. In: 26th Annual Symposion On Foundations of Computer Science. IEEE, The Institute of Electrical and Electronics Engineers, IEEE Computer Society Press, pp. 496–504.

Siegrist, K., 2001. Virtual Laboratories in Probability and Statistics. University of Alabama, Ch. C.4, Finite Sampling Models: The Multivariate Hypergeometric Distribution.
URL http://www.math.uah.edu/stat/urn/urn4.html

Stadlober, E., Zechner, H., Jan. 1999. The patchwork rejection technique for sampling from unimodal distributions. ACM Trans. Modeling and Comp. Simul. 9 (1), 59–80.

Valiant, L. G., 1990. A bridging model for parallel computation. Communications of the ACM 33 (8), 103–111.

Vitter, J. S., 2001. External memory algorithms and data structures: Dealing with massive data. ACM Computing Surveys 33 (2), 209–271.

Vitter, J. S., Aggarwal, A., 1988. The input/output complexity of sorting and related problems. Communications of the ACM 31 (9), 1116–1127.

Wu, T., 1992. Computation of the multivariate hypergeometric distribution function. Computing Science and Statistics 24, 25–28.

Zechner, H., 1997. Efficient sampling from continuous and discrete unimodal distributions. Ph.D. thesis, Technical University Graz, Austria.

Zechner, H., Stadlober, E., 1993. Generating beta variates via patchwork rejection. Computing 50 (1), 1–18.