Randomized Permutations in a Coarse Grained Parallel Environment

Jens Gustedt

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

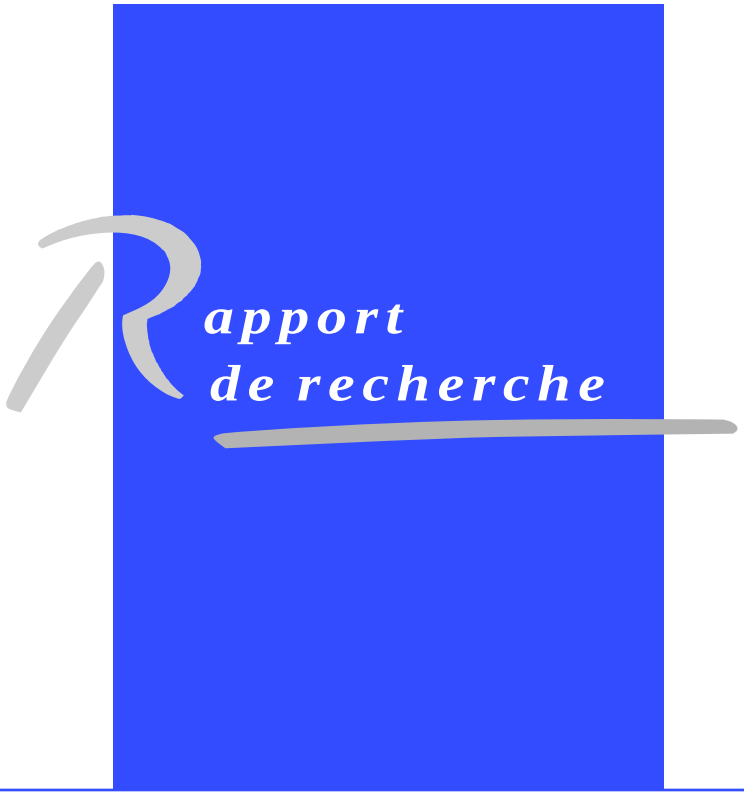
Randomized Permutations in a Coarse Grained Parallel Environment

Jens Gustedt

No 4639

Novembre 2002

THÈME 1



ISSN 0249-6399 ISRN INRIA/RR--4639--FR+ENG

Randomized Permutations in a Coarse Grained Parallel Environment

Jens Gustedt*

Thème 1 — Réseaux et systèmes
Projet Résédas

Rapport de recherche n˚4639 — Novembre 2002 — 12 pages

Abstract: We show how to distribute data at random (not to be confounded with permutation routing) in a coarse grained parallel environment with p processors. Previously known methods were not 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 give algorithms to compute it efficiently.

Key-words: randomization, permutations, coarse grained parallelism, PRO

(Résumé : tsvp)

* INRIA Lorraine / LORIA, France. Email: Jens.Gustedt@loria.fr – Phone: +33 3 83 59 30 90

Unité de recherche INRIA Lorraine
Technopôle de Nancy-Brabois, Campus scientifi que,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY (France)
Téléphone : 03 83 59 30 30 - International : +33 3 3 83 59 30 30
Télécopie : 03 83 27 83 19 - International : +33 3 83 27 83 19
Antenne de Metz, technopôle de Metz 2000, 4 rue Marconi, 55070 METZ
Téléphone : 03 87 20 35 00 - International: +33 3 87 20 35 00
Télécopie : 03 87 76 39 77 - International : +33 3 87 76 39 77

Permutations randomisés dans un environnement parallèle à gros grain

Résumé : Nous montrons comment distribuer des données aléatoirement dans un environnement parallèle à gros grain. Les méthodes qui étaient connus avant ne suffisait pas simultanément au trois critères de l'uniformité, de l'optimalité en travail et de la balance entre les processeurs. Pour garantir l'uniformité nous étudions la matrice de communication entre processeurs. Nous montrons que cette matrice est une généralisation de la distribution hypergéométrique multivariable et nous donnons des algorithmes pour la calculer efficacement.

Mots-clé : randomisation, permutations, parallèlisme à gros grain, PRO

1 Introduction and Overview

Random permutation 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 100 clock cycles on commonly used architectures such as a 300 MHz Sparc or a 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. Another important issue is that a standard algorithm to generate random permutations addresses memory in an unpredictable way and thus causing a lot of cache misses. The running time of a permutation program is more or less bound to the cpu-memory bandwidth. In the above tests this bottleneck amounts to about 33% (Sparc) and 80% (Pentium) of the wall clock time.

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 compute a random permutation in a parallel or distributed setting. But it also might have (at least long term) implications on the sequential framework.

When designing an alternative to the straight forward random generation of permutations, 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 items.

PRO only considers algorithms that are both time- and space-optimal. As a consequence of this enforced optimality, a PRO-algorithm always yields linear speed-up relative to a reference sequential algorithm. In addition, PRO assumes that the coarse grained communication cost only depends on p and 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 for the PRAM setting (see eg Reif [1985], Hagerup [1991]) and which assumptions are not too realistic. Please also note, that the so-called *permutation routing* problem (see eg 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 which uses general sorting as a subroutine. By that this algorithm has a superlinear total cost ($\log n$ per item) and is not work-optimal. Guérin Lassous and Thierry [2000] investigated several other algorithms to compute 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. It is not even to see that all permutations can be obtained nor is it in general possible to prove uniformity.

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 : 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, computation time, random numbers and bandwidth.*

The organization of this paper is as follows. Section 2 introduces the idea of separating the sampling of a 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 concludes by briefly discussing some experiments and tests.

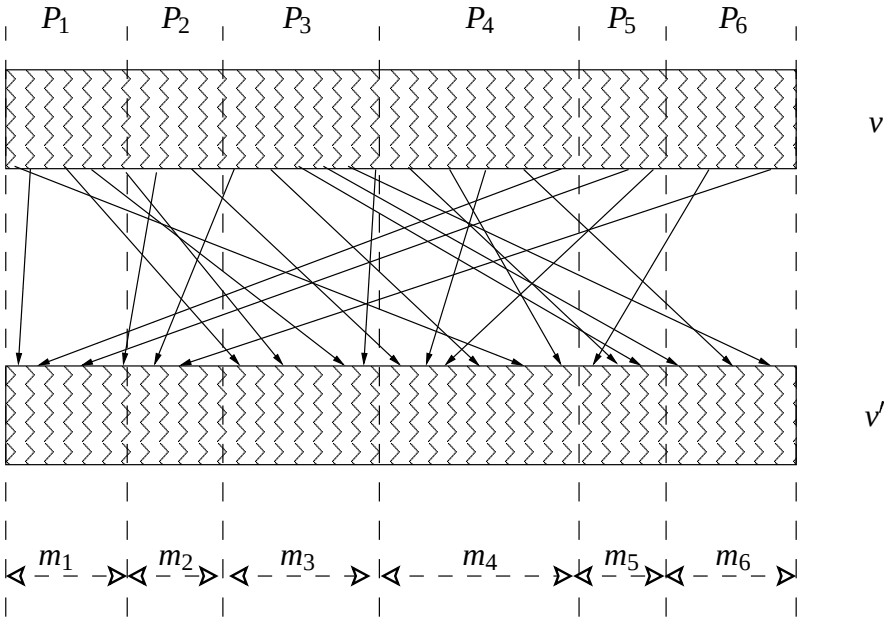


FIG. 1 – A vector an a permuted copy distributed on 6 processors

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

Given a vector v of size n our goal is to compute 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. Figure 1 tries to visualize such a situation. 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-blocks B_i that send the data to p' target-blocks B'_i such that the block 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 in Blocks)

Input *: Vector v of n items in total that is distributed on p processors such that processor P_i holds a block B_i of m_i elements ; target vector v' of length n such that P'_i holds a block B'_i of m'_i 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 the amount of block B_i to block B'_j communication. A is not an arbitrary matrix but has special properties. We have

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

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

A permutation can then be realized by the 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 send 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 Algorithm 1.

Algorithm 1: Parallel Random Permutation

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

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

foreach P_i, $i = 0, \ldots, p-1$ **do** permute B_i locally in parallel
Choose $A = (a_{i,j})$ according to (2) and (3)
foreach P_i, $i = 0, \ldots, p-1$ **do**
  for $j = 0, \ldots, p'-1$ **do** send $a_{i,j}$ items to processor P'_j
foreach P'_j, $j = 0, \ldots, p'-1$ **do**
  for $i = 0, \ldots, p-1$ **do** receive $a_{i,j}$ items from processor P_i
foreach P'_j, $j = 0, \ldots, p'-1$ **do** permute B'_i locally in parallel

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 *Algorithm 1 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 quiet 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. □

If we want a uniform distribution not all matrices A occur with the same probability. The distribution is not trivial and in particular the different a_{ij} are not independent. In the main part of the paper we will see 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 doesn't dominates the running time of Algorithm 1.

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

Proposition 2 *If the communication matrix A in Algorithm 1 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. □

Theorem 2 *There is a PRO-algorithm for computing a matrix as described in Problem 2 that has an optimal grain :*

– *There is a sequential algorithm that samples such a matrix with linear cost, i.e $O(p^2)$.*
– *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*. It has the probability

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

The hypergeometric distribution can be sampled quite efficiently, see Zechner [1994]. For the experiments that will be described in Section 6 the amount of random numbers per sample of $h(,,)$ was always less than 1.5 on average and 10 for the worst case.

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 B_i as being "white", $w=m_j$, 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 B'_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 eg. Siegrist [2001] for the terms.

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

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 $A^*=(a^*_{r,s})$ for*

$$a^*_{r,s}=\sum_{\substack{i_r<i\le i_{r+1}\\ j_s<j\le j_{s+1}}} a_{i,j}$$

is distributed as for the problem with input blocks of size $m^*_r=\sum_{i_r<i\le i_{r+1}} m_i$ and $m^{*'}_s=\sum_{j_s<j\le j_{s+1}} m'_j$.

Proof: This follows directly by the fact that we may join the input blocks B_i and output blocks B'_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 $A^*=(a^*_{r,s})$ be as above. Then each $a^*_{r,s}$ is distributed with $h(t,w,b)$ with*

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

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 2 different processors. Once 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 $s = 0, \ldots, p'$. Suppose $(a^*_{1,s})$ 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 compute 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 (Algorithm 2) 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 Algorithm 3.

Algorithm 2: Sequential sampling of a multivariate hypergeometric distribution

Input: Integers p and m, a vector of p values (m'_i)
$m \leq n = \sum_{i=0,\ldots,p-1} m'_i$.

Output: Random vector (α_i) distributed with a multivariate hypergeometric distribution with parameters m and (m'_i).

for *i=0,..,p-1* **do**
  Choose `toRight` according to $h(m, n - m'_i, m'_i)$
  Set $n = n - m'_i$, $\alpha_i = m -$ `toRight` and $m =$ `toRight`.

Algorithm 3: 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-1,..,0* **do**
split Choose vector $(\texttt{toUp}_j)$ according to a multivariate hypergeometric distribution with parameters m_i and (m'_j)
update **for** *j=0,..,p'-1* **do** $a_{i,j} = m'_j -$ `toUp`$_j$ and $m'_j =$ `toUp`$_j$

Proposition 7 *Algorithm 3 computes $(a_{i,j})$ with $O(p \cdot p')$ basic operations and $O(p \cdot p')$ calls to $h(,,)$.*

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 Algorithm 2 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 Algorithm 4. Observe that Algorithm 2 could be seen as an iterative variant of Algorithm 4 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. In practice this may speed up this particular part of the computation quite efficiently.

Algorithm 4: `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** $(m'_j)_{j=0,\ldots,p'}$ **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 $(a_{i,.})_{i=0,\ldots,q-1}$ with `RecMat`$(q,(m_i),p',(\texttt{toLo}))$
- Sample $(a_{i,.})_{i=q,\ldots,p-1}$ with `RecMat`$(p-q,(m_{i+q}),p',(\texttt{toUp}))$

5 Parallel algorithms

We will be derive our first parallel algorithm from Algorithm 4 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.

Algorithm 5: Parallel sampling of a communication matrix (with a log-factor in the total work)

Input: Total number of processors p, processor `id` with $0 \le \texttt{id} < p$ and value M.

Result: For each processor P_i a vector (β_i) 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.

if $id = 0$ **then** Initialize $(\beta_i) \equiv M$

$r = 0$ and $s = p$

while $(s - r) > 1$ **do**

- $q = (r+s)/2$
- **switch** id **do**
  - **case** r :
    - $t = (s-q)M$
    - Choose vector $(\texttt{toUp}_i)$ according to a multivariant hypergeometric distribution with parameters t and (β_i)
    - Send $(\texttt{toUp}_i)$ to processor P_q
    - $(\beta_i) = (\beta_i - \texttt{toUp}_i)$
  - **case** q : Receive (β_i) from processor P_r
- **if** $id < q$ **then** $r = q$ **else** $s = q$

Proposition 8 *Algorithm 5 is correct. It has running time, communication and calls to $h(,,)$ of $\Theta(p \log p)$ per processor and a total work, communication and calls to $h(,,)$ of $\Theta(p^2 \log p)$.*

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 Algorithm 4 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 the matrix generation. We have $\log p$-factor both in the time as 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 Algorithm 5 we are only a log-factor away from the optimal granularity of $\sqrt{n}$.

Algorithm 6: Cost-optimal parallel sampling of a communication matrix

Input: Total number of processors p, processor `id` with $0 \leq$ `id` $< p$ 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` $= 0$ **then** Initialize (β_i^0) and (β_i^1) to M
$\Delta = 0$ and $\nabla = 1$
Set r, r_0, r_1 to 0 and s, s_0, s_1 to p

2 **while** $(s - r) > 1$ **do**
  $q = (r+s)/2$ and $q_\Delta = (r_\Delta + s_\Delta)/2$
  switch `id` **do**
    case r *:*
      for *the range* $q_\Delta \leq i < s_\Delta$ **do**
        $t = \sum_{q_\Delta \leq i < s_\Delta} \beta_i^\Delta$
        Send all these β_i^Δ to processor P_q
      for *the range* $r_\nabla \leq i < s_\nabla$ **do**
        Choose vector $(\texttt{toUp}_i)$ according to a multivariant hypergeometric distribution with parameters t and (β_i^∇)
        Send all these $\texttt{toUp}_i$ to processor P_q
        Set all the $\beta_i^\nabla = \beta_i^\nabla - \texttt{toUp}_i$
    case q *:*
      for *the range* $q_\Delta \leq i < s_\Delta$ **do** Receive all β_i^Δ from P_r
      for *the range* $r_\nabla \leq i < s_\nabla$ **do** Receive all β_i^∇ from P_r
  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 $r_0 \leq i < s_0$ and $r_1 \leq j < s_1$ according to the input vectors $(\beta_i^0)_{r_0 \leq i < s_0}$ and $(\beta_i^1)_{r_1 \leq i < s_1}$.

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

Proposition 9 *Algorithm 6 is correct. It has running time, communication and calls to $h(,,)$ of $\Theta(p)$ per processor and a total work, communication and calls to $h(,,)$ of $\Theta(p^2)$.*

Proof: The proof is analogous to that of Proposition 8. The difference between the two algorithms is that the matrix is not only sliced in one dimension but evenly split along both dimension (controled by Δ and ∇). At the end of the **while**-loop every processor is left with a sub-matrix that it has to compute. The size of this submatrix is

$$O(2^{\frac{1}{2}\log p} \cdot 2^{\frac{1}{2}\log p}) = O(p) \tag{9}$$

This shows in particular that all processors only have to handle data in the size of $O(p)$.

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

6 Outlook

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 48 processors and 480 million items. The framework for the implementation was SSCRAP, see Essaïdi et al. [2002], an environment for coarse grained algorithms.

The overhead due to the parallelization over the simple sequential algorithm is a factor between 3 and 5 as one would expect : we have to perform two local permutations and the communication between the processors. Typical running times for 480 million elements on a 400 MHz Origin are : 137 sec (sequential), 210 sec (3 proc), 107 sec (6 proc), 72.9 sec (12 proc), 60.9 sec (24 proc) and 53.2 sec (48 proc). The description of the implementation and the tests will be subject of a paper by its own.

The tests showed that the main limitation for Algorithm 1 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 can be a bottleneck. So in situations where medium sized permutations are needed repeatedly a parallel implementation of the matrix sampling will be helpful.

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], and as the gap between CPU and memory speed is constantly growing there is also hope that the parallel algorithms can give rise to sequential algorithms and implementations that avoid part of the cache misses of the straight forward algorithm.

Acknowledgement

The author thanks Kjell Kongsvik and Mohamed Essaïdi for their help with the implementation.

Références

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

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

F. Dehne, W. Dittrich, and D. Hutchinson. Effi cient external memory algorithms by simulating coarsegrained parallel algorithms. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 106–115, 1997.

Mohamed Essaïdi, Isabelle Guérin Lassous, and Jens Gustedt. SSCRAP : An environment for coarse grained algorithms. In *Fourteenth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002)*, 2002.

Assefaw Hadish Gebremedhin, Isabelle Guérin Lassous, Jens Gustedt, and Jan Arne Telle. PRO : a model for parallel resource-optimal computation. In *16th Annual International Symposium on High Performance Computing Systems and Applications*, pages 106–113. IEEE, The Institute of Electrical and Electronics Engineers, 2002.

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

Isabelle Guérin Lassous and Éric Thierry. Generating random permutations in the framework of parallel coarse grained models. In *Proceedings of OPODIS'2000*, volume 2 of *Studia Informatica Universalis*, pages 1–16, 2000.

Torben Hagerup. Fast parallel generation of random permutations. In *Automata, Languages and Programming*, volume 510 of *Lecture Notes in Comp. Sci.*, pages 405–416. Springer-Verlag, 1991. Proceedings of the 18th International Colloquium ICALP'91.

Clyde P. Kruskal, Larry Rudolph, and Marc Snir. A complexity theory of effi cient parallel algorithms. *Theoretical Computer Science*, 71(1) :95–132, march 1990.

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

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

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

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

H. Zechner. *Effi cient sampling from continuous and discrete unimodal distributions*. PhD thesis, Technical University Graz, Austria, 1994.

Unit´e de recherche INRIA Lorraine, Technopˆole de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhˆone-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399