GENERATING RANDOM PERMUTATIONS IN THE FRAMEWORK OF COARSE GRAINED MODELS

ISABELLE GUÉRIN LASSOUS AND ERIC THIERRY

Abstract. We present three algorithms for generating random permutations in the coarse grained model CGM. For each of the proposed algorithms, we study the number of supersteps, the size of the local memory, the overall communication cost and we check if it gives a permutation with the uniform distribution or not. The proposed algorithms are intended to be simple and of practical relevance; the difficulty, in this paper, lies in proving that they have the desired properties.

Keywords: coarse grained models, CGM, random permutations, algorithms

1. Introduction

Permutations often constitutes the basis to construct random combinatorial objects (lists, trees, graphs, etc) in computer science and thus are key data to test and experiment programs [Den94, FZC94, CC86, CF89].

Some algorithms require good distribution properties of the input in order to be efficient. Therefore it is interesting to have efficient algorithms that randomly permutes the data according to a good distribution. For instance, some randomized distributed algorithms number the objects and assume that in average for each object the numbers assigned to its neighbors are far enough from its own number. A preprocessing phase that randomly permutes the input data makes this assumption correct whatever the input. For an example, see [Sib97]. Note that random permutations of data is equivalent to random permutations of integers.

Few articles deal with the permutation of integers in parallel/distributed setting [Rei85, And90, AS96]. Moreover, all these articles propose algorithms

*I. Guérin Lassous, INRIA Rhône-Alpes, ENS Lyon, 46 alle d'Italie, 69364 LYON Cedex 07 - Email: Isabelle.Guerin-Lassous@ens-lyon.fr, Fax: +33 4 72 72 80 80 – corresponding author

†E. Thierry, LIRMM, Université de Montpellier, 34392 Montpellier Cedex 5 - Email: thierry@lirmm.fr - Fax : +33 4 67 41 85 00

Algorithms	Nb of supersteps	Size of local memory	Overall com. cost	Uniform distribution
1	1	$O(\frac{n}{p})$ (high proba.)	$O(n)$	yes
2	2	$O(\frac{n}{p})$	$O(n)$	no
3	$2\log_2 p$	$O(\frac{n}{p})$	$O(n \log p)$	yes

TABLE 1. Overview on the proposed algorithms (n is the input size and p is the number of processors)

in the classical PRAM model [Jàj92]. But this model is far from being realistic because it assumes that $p = \theta(n^k)$ and that any processor can exchange data with any other processor at constant cost. In this paper, we consider more realistic models that are the coarse grained models like BSP, LogP or CGM [Val90, CKP$^+$93, DFRC96]. These models are devoted to the design of distributed algorithms by reflecting constraints of real machines without being specific to a peculiar machine. They have been developped to facilitate the implementations of such algorithms and lead to efficient and portable code.

In this paper, we address the problem of the random permutation of integers in the framework of coarse models. Our aim is to design algorithms with a small number of supersteps (superstep is defined in Section 2) and a small overhead in communication cost, without overloading the local memories and with a specified random distribution on the output. We propose three different algorithms. For each algorithm, we consider the number of supersteps, the size of the used local memory of each processor, the overall communication cost and we check if the algorithm gives a permutation with the uniform distribution or not. Table 1 shows these features for each proposed algorithm.

In Section 2, we quickly describe the coarse grained models and especially CGM (Coarse Grained Multicomputer). In Section 3, we present Algorithm 1 which is based on the choice of one processor for each element. Then we give Algorithm 2 that uses the division by packets of equal size. Finally we show Algorithm 3 that requires only point-to-point communications.

Note, that all the algorithms that are presented are intended to be simple, but that the difficulty lies in proving that they have the desired properties.

2. The Coarse Grained Models

The coarse grained models assume that p the number of processors is small with respect to the input size n. This asumption includes many existing machines. Valiant is the first to have proposed such a model with BSP [Val90]. Some variations of BSP have been developped like LogP and CGM [CKP$^+$93, DFRC93]. In this paper, we chose to work on the simplest model (that is

CGM) because we think that it is sufficient to design efficient algorithms for the random permutation of integers. Moreover, it has be shown in [GLGM00] that algorithms written in CGM turn out to be feasible, efficient, predictable and portable in most cases.

CGM assumes that: 1) all processors execute the same program, 2) each processor holds $O(\frac{n}{p})$ data and 3) an algorithm is an alternation of local computations and global communications. A phase of local computations followed by a global communication is called a *superstep*.

When designing an algorithm in this model, the goal is to have a small number of local computations, of supersteps and a small overhead in communication costs. It is usually assumed that the number of supersteps does not have to depend on the input size n because it would lead to poor performances.

All of our algorithms use arrays to maintain their data. To explain the algorithms, we often assume that the data are put in a *global array* and that each processor works on a part of this array. In fact, each processor has a *local array* and the union of these arrays forms the global array (which is virtual). We give now an example of a CGM algorithm that we will use in the following. This algorithm re-balances the input such that each processor stores exactly $\frac{n}{p}$ data.

Algorithm 0: Balance

Input: array T of n elements, n_i data per processor i $(1 \leq i \leq p)$

Output: array T of n elements evenly distributed among the processors

1 each processor i sends n_i to all the processors j such that $j > i$

2 each processor i computes the global index of its elements (equal to the local index $+n_1 + .. + n_{(i-1)}$)

foreach *processor* i **do**

3   **for** *(*$j = 1; j \leq n_i; j = j+1$*)* **do**

        send element $T[j]$ and its new position ($global\ index\ of\ T[j]$ mod $(\frac{n}{p}+1)$) to the processor $\lfloor \frac{global\ index\ of\ T[j]}{\frac{n}{p}+1} \rfloor + 1$

each processor i receives its data

Note that Algorithm 0 has exactly two communication steps (Steps 1 and 3) and requires $O(p + \frac{n}{p})$ local computations at Step 2. Note that at Step 2 we assume that $p \leq \frac{n}{p}$. The overall communication cost is $O(n)$.

Henceforth, we will number input data with integers 1 to n where n is the number of data we want to permute. Our problem thus consists of permuting these integers with the constraints exposed above (features 1), 2) and 3) of CGM and, in addition a minimal number of supersteps).

3. Independent choice of processors

The idea of Algorithm 1 is to adapt the simple PRAM algorithm proposed in [Rei85] to the coarse grained models: each processor independently chooses for each of its elements a target processor and sends it to this processor.

Algorithm 1: Independent choice of processors

Input: array T of n elements divided evenly among the p processors
Output: the random permutation of the data of T stored in an array P

each processor randomly chooses an integer between 1 and p for each element of T
1 **for** $(j = 1; j \leq p; j = j + 1)$ **do**
    Each $T[i]$ that is associated to the integer j is sent to processor j

the received values are stored in an array P
2 each processor locally permutes at random the data of P

Proposition 1. *Algorithm 1 randomly permutes the input data according to the uniform distribution in $O(\frac{n}{p})$ local computations with high probability and in one communication step. The overall communication cost is $O(n)$.*

Proof. At Step 2, each processor permutes its data with any sequential algorithm generating permutations with the uniform distribution [Knu81]. It is easy to see that Algorithm 1 requires only one communication step of total size $O(n)$ at Step 1 and for each processor the local computations are linear in the size of its data.

During the execution of Algorithm 1, it is possible that the data are not equally distributed among the processors so that the memory of some processors might be overloaded compared to what is allowed by the chosen model. Nevertheless, it is easy to balance the data equally likely among the processors by using Algorithm 0. By a similar proof as in [Rei85], it follows that with very high probability the number of elements of P in each processor is smaller than $O(\frac{n}{p})$: $|P|_k$, $1 \leq k \leq p$, denotes the number of elements of P on the processor k. There exists a constant c such that:

$$\forall \alpha \geq 1, Prob(|P|_k \leq c\alpha \frac{n}{p}) \geq 1 - \frac{1}{e^{\alpha \frac{n}{p}}} \geq 1 - \frac{1}{e^{\frac{n}{p}}}. \tag{1}$$

In order to prove the uniform distribution on permutations, we study the reverse construction of Algorithm 1: we start from any permutation that we randomly cut in p packets (some packets may be empty). Now we know the initial value v ($1 \leq v \leq p$) given to each element of T. It means that this cut is associated to a unique sequence of random samples at Step 1. This construction is independent of the starting permutation, therefore the distribution of permutations generated by Algorithm 1 is uniform. □

4. Division by packets of equal size

4.1. **The Algorithm.** Unlike Algorithm 1, Algorithm 2 ensures that the data are always equally distributed among the processors during its execution. Without loss of generality, we can assume that p divides $\frac{n}{p}$.

Algorithm 2: Division by packets

Input: array T of n elements divided evenly among the p processors
Output: the random permutation of the data of T stored in an array P

1 each processor locally permutes the data of T in a random way
each processor divides T in p packets of size $\frac{n}{p^2}$
2 **for** $(j = 1; j \leq p; j = j + 1)$ **do**
  each processor sends the packet j to processor j
3 each processor receives the distinct packet and puts them in P
each processor locally permutes at random the data of P

It is easy to see that the data size on one processor is fixed during the execution of Algorithm 2 and is equal to $\frac{n}{p}$. This algorithm requires also only one communication step of total size $O(n)$ at Step 2.

The drawback of Algorithm 2 is that the generated permutations have not a uniform distribution: at the end of this algorithm, we know that each processor has $\frac{n}{p^2}$ elements numbered between $(i-1)\frac{n}{p}$ and $i\frac{n}{p} - 1$ $(1 \leq i \leq p)$. Algorithm 2 can not generate all the permutations if we run it once. However it is interesting to study what is happening when we iterate this algorithm, as shown in Section 4.2.

Note that [LS94] and [BHJ96] deal with the routing problem according to a given permutation but their context and their aim are different from ours: the target machine of [LS94] is fine grained and we do not think that their results can be extended to the coarse grained models; using the algorithm of [BHJ96] may lead to cases where some processors hold more than $\frac{n}{p}$ data.

4.2. **Iteration of Algorithm 2 .**

Proposition 2. *Any permutation can be obtained after exactly two iterations of Algorithm 2.*

Proof. Since Algorithm 2 starts and ends with local permutations of the elements on each processor, we don't need to consider the order of the elements in the arrays. So we reason on packets of elements on each processor. We assume $n = kp^2$ where k is an integer.

The set of elements on a processor after the first application of Algorithm 2 will be called *transition packet*. The problem is to route the elements from the source packets to the target packets through the transition packets, with respect to the rules of Algorithm 2. In other words, what we need is to show that there exists a configuration of p transition packets where each transition packet holds exactly k elements of each source packet as well as k elements of each target packet, so that it corresponds to two iterations of Algorithm 2.

To show the existence of such a configuration fulfilling the properties given above, we build the following network:

- a vertex for each source packet, a vertex for each target packet, a "source" vertex and a "target" vertex,
- an edge with weight 1 between the "source" and each source packet,
- an edge with weight 1 between each target packet and the "target",
- an edge between each source packet and each target packet weighted by the number of elements going from the source packet to the target packet.

Figure 1 shows a configuration of source and target packets and the associated network.

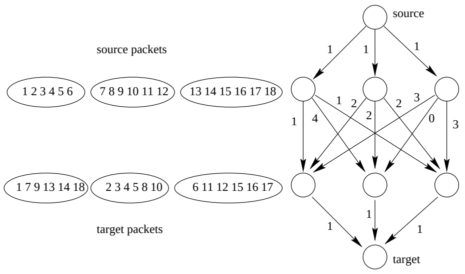

FIGURE 1. An example of network associated to a set of target packets

We now prove the existence of a "perfect matching": there exists a set of p elements composed of exactly one element of each source packet such that this set is also divided into one element by target packet.

The network is such that all the sums of the weights of the edges starting from source packets and all the sums of the weights of the edges ending at target packets are equal (to $\frac{n}{p}$). As a consequence, a flow of weight p actually exists: we use the min-max theorem which shows that the maximal flow is equal to the minimal cut and the computation of the minimal cut gives p (see the annex for the details of the computation) [CLR90]. Thus we choose a flow of weight p and for each edge between a source packet and a target packet whose flow is 1, we arbitrarily choose an element belonging to the source packet and the target packet. We keep this set of p elements and remove these elements from the source and target packets.

With the remaining elements, we can build the same network as above. This network verifies the same property concerning the sums of weights. Thus we can remove again a set of p elements evenly distributed among the p source packets and among the p target packets. We repeat the process until there is no element left in the packets.

At the end, we have pk sets of p elements such that each set is composed of one element per source packet and is also evenly distributed among the target packets. Then we arbitrarily gather these sets by groups of k sets in order to obtain p transition packets of pk elements. These transition packets verify the properties we expected: they receive exactly k elements from each source packet and they send exactly k elements towards each target packet. So it is possible to reach the target packets from the source packets after two iterations of Algorithm 2. □

Figure 2 shows an example of network, a sequence of flows, the associated sets and two iterations of Algorithm 2 which lead to the target packets.

4.3. Convergence of the distribution.

Proposition 3. *Whatever the initial distribution, by iterating Algorithm 2 the limit of the distribution is the uniform distribution.*

Proof. Algorithm 2 transforms any initial permutation of n elements into another one with the following properties:

- given the initial permutation, two different sequences of random samples lead to two different permutations,
- therefore, as the random samples have the same probability, all the permutations which can be generated from an initial permutation are equally likely to appear. By symmetry, this probability is the same for any initial permutation,
- if a permutation can be transformed into another one, the second one can be inversely transformed into the first one.

The random transformation of permutations can be seen as a Markov process: the states are the permutations and the directed graph of transitions corresponds to the possible transformations from a permutation into another one with the associated probability. Considering the three previous remarks,

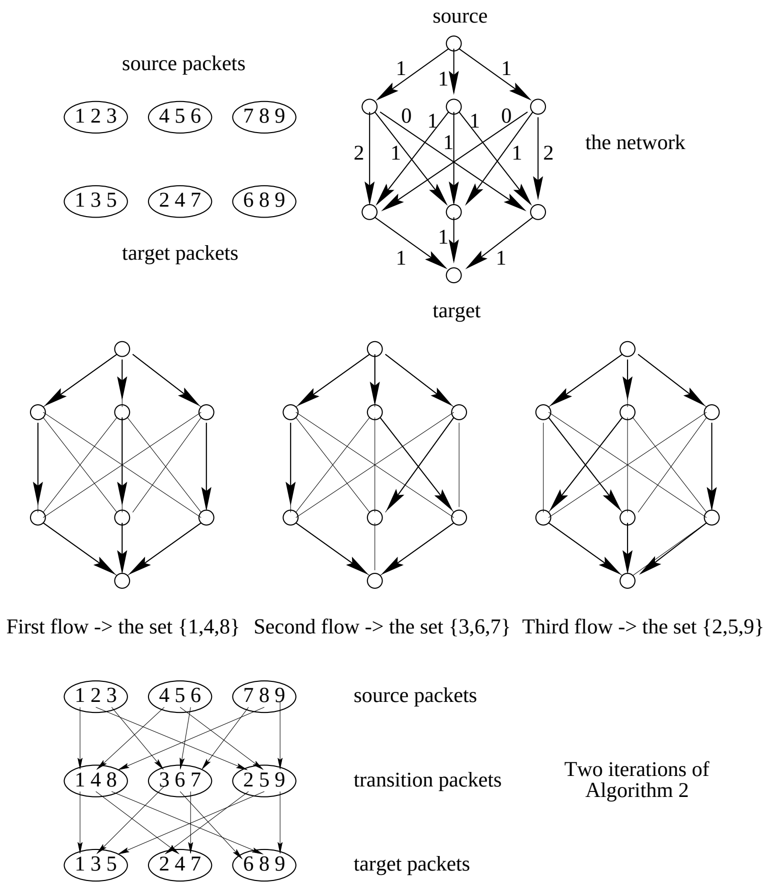

FIGURE 2. The construction of transition packets, in order to route the elements to the target packets.

it is a symmetric regular graph, of degree r ($r = \dfrac{((kp)!)^{2p}}{(k!)^{p^p}}$) and where all the edges have the same weight ($1/r$). As this graph is symmetric, it can be considered as an undirected graph.

Let A be the adjacency matrix of the graph of the transformation between permutations. Let $M = [m_{i,j}]$ be the associated Markov matrix, i.e. defined by $M = \frac{1}{r}A$:

$$m_{i,j} = \begin{cases} 1/r & \text{if and only if } a_{i,j} = 1 \\ 0 & \text{otherwise} \end{cases}$$

If π_t is a distribution on the $n!$ permutations (a vector with $n!$ non-negative coefficients of sum 1), then after applying Algorithm 2, the new distribution π_{t+1} verifies: $\pi_{t+1} = M\pi_t$.

Then we can use the following theorem about the convergence of finite Markov chains:

Theorem 1. [Nor97] *If the transition matrix M is such that at least one of its powers has no coefficient equal to zero, then when $t \longrightarrow \infty$, $\pi_t \longrightarrow \pi$ for any initial distribution π_0. This limit π is the unique stationary distribution of the Markov chain, i.e. verifying $\pi = M\pi$.*

This theorem applies to our transformation: as a consequence of Proposition 2, all the coefficients of the matrix M^2 are positive (there is always a path of length 2 in the graph linking two permutations). It remains to solve the matricial equation $\pi = M\pi$, which is equivalent to $r\pi = A\pi$.

It is known that, the adjacency matrix A of any regular graph of degree r verifies the following properties:

- The eigenvectors associated to the eigenvalue r belong to the line generated by vector $(1, 1, \ldots, 1)$. So π is the uniform distribution.
- The other eigenvalues of A have absolute values strictly less than r.

$\square$

5. Point-to-point communications

In order to ensure that the memory of each processor can not get overloaded, another solution consists in following a special rule: during a communication step, a processor sends data to only one processor and receive data from only one processor. We now describe an algorithm based on this rule. Without loss of generality, we assume that $p = 2^k$.

The arrows on Figure 3 show which pairs of processors exchange data at each round when $p = 8$.

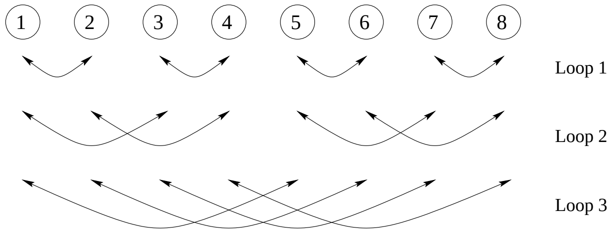

Figure 3. Example of the execution of Algorithm 3.

Algorithm 3: Point-to-point communications

Input: array T of n elements evenly divided among the p processors
Output: the random permutation of the data of T

for *(* $i = 1; i < k + 1; i = i + 1$ *)* **do**
  foreach *processor num* **do**
    if $\lceil \frac{num}{2^{i-1}} \rceil \pmod 2 = 1$ **then**
1   choose equally likely at random a subset of the elements of T
      count the number $size$ of elements of this subset
      send $size$ to processor $num + 2^{i-1}$
      send the chosen subset of T to processor $num + 2^{i-1}$
      receive the data sent by the processor $num + 2^{i-1}$
    else
      receive the value $size$ sent by the processor $num - 2^{i-1}$
2   choose a random subset of its elements of size $size$ with equiprobability
      send this chosen subset of T to processor $num - 2^{i-1}$
      receive the data sent by the processor $num - 2^{i-1}$

Proposition 4. *Algorithm 3 randomly permutes the input data according to the uniform distribution in* $O(\frac{n}{p})$ *local computations per superstep and in* $2\log_2 p$ *communication steps. The overall communication cost is in* $O(n \log p)$.

Proof. At Step 1, in each initiator processor (processor having $work$ equal to $true$), each element of T is chosen with probability $\frac{1}{2}$, which constitutes a random subset of T. Step 2 can be realized with a local permutation on the elements of T and then by taking the $size$ first elements of T.

It is straightforward to show that Algorithm 3 requires $2\log_2 p$ communication steps and $O(\frac{n}{p})$ local computations per superstep. Each step communicates $O(n)$ data, then the overall communication cost is in $O(n \log p)$.

Algorithm 3 generates permutations with uniform distribution: all the sequences of exchanges of sets have the same probability, and given a permutation there exists one and only one sequence of exchanges which leads to this permutation. □

Note that, if $p \leq \frac{n}{p}$, then Algorithm 3 requires at least $\log_2 p$ supersteps (as shown in Proposition 5). The assumption that $p \leq \frac{n}{p}$ is a usual hypothesis in the coarse grained models, where we often assume that each processor can at least hold the table of all the processors.

Proposition 5. *If $p \leq \frac{n}{p}$, an algorithm which generates all the permutations with the same probability and respects the rule of point-to-point communications requires at least $\log_2 p$ communication steps.*

Proof. Consider p elements in the first processor. As the algorithm is supposed to generate any permutation, it should be able to generate a permutation such that each of the p elements is allocated to a different processor. Owing to the rules we follow at each round of communication, the elements of a processor are either left on this processor or sent to a unique other processor. As a consequence, after k steps the elements of one processor are spread over at most 2^k different processors. It means that the algorithm needs at least $\log_2 p$ steps of communication to send the p elements over the p processors. □

6. Conclusion

We have presented three distributed algorithms for generating random permutations in the coarse grained models setting. Note that none of these algorithms succeeds in generating permutation with the uniexactly two iterations of Algorithm 2form distribution in a constant number of supersteps and without exceeding $O(\frac{n}{p})$ data per processor.

It will be interesting either to design such an algorithm with these properties, or to show that it is impossible to obtain such an algorithm with such constraints.

We are currently working on the implementation side of these algorithms in order to show they are practically relevant.

References

[And90] R. Anderson. Parallel algorithms for generating random permutations on a shared memory machine. In *Proc. SPAA'90*, pages 95–102, 1990.

[AS96] L. Alonso and R. Schott. A parallel algorithm for the generation of permutation and applications. *Theoretical Computer Science*, 159(1):15–28, 1996.

[BHJ96] D. A. Bader, D. R. Helman, and J. Jj. Practical Parallel Algorithms for Personalized Communication and Integer Sorting. *ACM Journal of Experimental Algorithmics*, 1, 1996.

[CC86] B. Chan and M. S. Chern. Parallel generation of permutations and combinaisons. *BIT*, 26:277–283, 1986.

[CF89] M. Cosnard and A. Ferreira. Generating permutations on a VLSI suitable linear network. *The Computer Journal*, 32(6):571–573, 1989.

[CKP+93] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceeding of 4-th ACM SIGPLAN Symp. on Principles and Practises of Parallel Programming*, pages 1–12, 1993.

[CLR90] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[Den94] A. Denise. *Méthodes de génération d'objets combinatoires de grande taille et problèmes d'énumération.* PhD thesis, Université de Bordeaux I, 1994.

[DFRC93] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable Parallel Geometric Algorithms for Coarse Grained Multicomputer. In *ACM 9th Symposium on Computational Geometry*, pages 298–307, 1993.

[DFRC96] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. International Journal on Computational Geometry, Vol. 6(No. 3):pp 379–400, 1996.

[FZC94] P. Flajolet, P. Zimmermann, and B. V. Custem. A calculus for the random generation of combinatorial structures. *Theoretical Computer Science*, 132:1–35, 1994.

[GLGM00] I. Guérin Lassous, J. Gustedt, and M. Morvan. Feasibility, Portability, Predictability and Efficiency: Four Ambitious Goals for the Design and Implementation of Parallel Coarse Grained Graph Algorithms. Technical Report RR-3885, INRIA Lorraine, 2000. http://www.inria.fr/RRRT/publications-fra.html.

[Jàj92] J. Jàjà. *An Introduction to Parallel Algorithm.* Addison Wesley, 1992.

[Knu81] D. E. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, 1981.

[LS94] J. Y-T. Leung and S. M. Shende. On Multidimensional Packet Routing for Meshes with Buses. *Journal of Parallel and Distributed Computing*, 20:187–197, 1994.

[Nor97] J. R. Norris. *Markov chains.* Cambridge University Press, 1997.

[Rei85] J. H. Reif. An optimal parallel algorithm for integer sorting. In *26th Symposium on Foundations of Computer Science*, pages 496–503, 1985.

[Sib97] J. F. Sibeyn. Better Trade-offs for Parallel List Ranking. In *Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures (SPAA'97)*, pages 221–230, 1997.

[Val90] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, Vol. 33(8):103–111, 1990.

Appendix

Probability of overloading a processor with Algorithm 1. Given a fixed processor, for instance processor 1, for any element, the probability to be sent to processor 1 is $1/p$. Therefore the probability that processor 1 receives more than k elements corresponds to the queue of the binomial distribution with parameter $1/p$:

$$\begin{aligned} Pr(|P|_1 \geq k) &= \sum_{i=k}^{n} C_n^i \frac{1}{p^i}(1-\frac{1}{p})^{n-i} \\ &\leq C_n^k \frac{1}{p^k} \end{aligned}$$

Thanks to the Stirling approximation formula, we also have the following upper bound:

$$C_n^k \leq (\frac{en}{k})^k$$

In our problem, $k = \alpha \frac{n}{p}$, with $\alpha \geq 1$. It gives us the upper bound:

$$Pr(|P|_1 \geq \alpha \frac{n}{p}) \leq \frac{1}{e^{\alpha(\log \alpha - 1)\frac{n}{p}}}$$

Computation of the minimal cut of the network built in section 3.1. For the basic definitions on flow networks, see [CLR90].

We suppose that the sums of the weights of the edges starting at each source packet and the sums of the weights of the edges ending at each source are all equal to d, $d \geq 1$. Details of the computation:

- considering the cut between the "source" vertex and the other vertices, the sum of the weights of the edges is p.
- given any cut as the one drawn on Figure 4, we add the weights of edges between the "source" vertex s and the vertices in A, between the vertices in B' and the "target" vertex t, between the vertices in A' and the vertices in B. The sum gives:

$$\begin{aligned} sum &= \sum_{a \in A} weight(s,a) + \sum_{b \in B'} weight(b,t) + \sum_{a \in A', b \in B} weight(a,b) \\ &= |A| + |B'| + \sum_{a \in A', b \in B} weight(a,b) \end{aligned}$$

- If $|A'| \leq |B'|$,

$$sum \geq |A| + |B'| \geq |A| + |A'| = p$$

- If $|A'| > |B'|$,

$$\begin{aligned}\sum_{a\in A',b\in B} weight(a,b) &= \sum_{a\in A',b\in B\cup B'} weight(a,b) - \sum_{a\in A',b\in B'} weight(a,b)\\ &\geq \sum_{a\in A',b\in B\cup B'} weight(a,b) - \sum_{a\in A\cup A',b\in B} weight(a,b)\\ &\geq |A'|d - |B'|d\end{aligned}$$

As a consequence:

$$\begin{aligned} sum &\geq |A| + |B'| + (|A'| - |B'|)d\\ &\geq |A| + |B'| + (|A'| - |B'|)\\ &\geq |A| + |A'| = p\end{aligned}$$

- Therefore the minimal cut is p

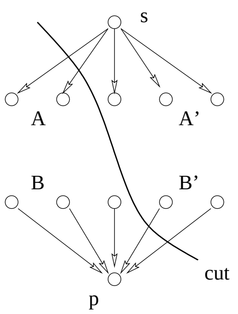

FIGURE 4. A cut of the network, the notations

Authors addresses:
I. Guérin Lassous, INRIA Rhône-Alpes, ENS Lyon, 46 alle d'Italie, 69364 LYON Cedex 07 - Email: Isabelle.Guerin-Lassous@ens-lyon.fr - Fax: +33 4 72 72 80 80
E. Thierry, LIRMM, Université de Montpellier, 34392 Montpellier Cedex 5 - Email: thierry@lirmm.fr - Fax : +33 4 67 41 85 00