Sublinear Communication for Integer Permutations

Jens Gustedt

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Sublinear Communication for Integer Permutations

Jens Gustedt

N° ????

december 2007

Thème NUM

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

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
NANCY - GRAND EST

Sublinear Communication for Integer Permutations

Jens Gustedt

Thème NUM — Systèmes numériques
Équipe-Projet AlGorille

Rapport de recherche n° ???? — december 2007 — 20 pages

Abstract: In [Gustedt (2007)] we have shown that random shuffling of data can be realised with linear resource usage, CPU time as well as communication, and this for a large variaty of paradigms, in particular distributed and out-of-core computation. In this paper we restrict to the case of permutations of the integers $[M] = \{1, \ldots, M\}$ and show first show how the communication for a p processor setting can be reduced from $O(M)$ words ($O(M \log M)$ bits, the coding size of the permutation) to $O(M \log p / \log M)$ words ($O(M \log p)$ bits, the coding size of a partition of $[M]$ into M/p sized subsets). For the common case of using pseudo-random numbers instead of real randomness, this coding size is in fact not a valid lower bound; we show the communication can be lowered to a use of bandwidth that is proportional to the used real randomness. The efficiency and feasibility of this second approach is demonstrated by large scale experiments where it proves its scalability and outperforms the previously known approaches by far. First, we compare our algorithm to the classical sequential data shuffle algorithm, where we get a speedup of about 1.5. Then, we show how the algorithm parallelizes well on a multicore system and scales to a cluster of 440 cores.

Key-words: integer permutations, parallel computing, random generation, pseudo-randomness, PRG, experiments, distributed computing

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

Communication sous-linéaire pour la génération de permutations d'entiers

Résumé : Avec [Gustedt (2007)] nous avons montré qu'une redistribution randomisée de données peut être réalisée avec une utilisation linéaire de ressources. Ceci aussi bien en ressources de calcul que de communication, et aussi dans une large variété de paradigmes, en particulier en calcul repartie et « *out of core* ». Ici nous restreignons cette approche aux permutations d'entiers $[M] = \{1, \ldots, M\}$ et nous montrons d'abord comment la communication entre processeurs peut être réduite de $O(M)$ mots (ou $O(M \log M)$ bit) à $O(M \log p / \log M)$ mots ($O(M \log p)$ bit) où p est le nombre de processeurs de la plate-forme. Ceci correspond à un codage d'une partition de $[M]$ en sous-ensembles de taille M/p. Pour le cas habituel d'utilisation de nombres pseudo-aléatoires au lieu de véritables aléas ce codage ne présente en effet pas une borne inférieure ; nous développons des outils qui réalisent un temps de communication qui est proportionnel à l'utilisation des bit aléatoires. La faisabilité et l'efficacité de cette deuxième approche sont démontrées par des expériences menées à large échelle, où notre algorithme fait preuve d'extensibilité et améliore largement les approches déjà connues : d'abord nous comparons notre algorithme à l'algorithme classique, qui est accéléré par un facteur de $1{,}5$ environ, puis nous montrons qu'il se parallèlise bien sur des systèmes à multi-cœurs et comment il s'étend à un cluster avec 440 cœurs.

Mots-clés : permutations d'entiers, calcul parallèle, génération aléatoire, nombres pseudo-aléatoires, expériences, calcul reparti

1 Introduction and Overview

The problem of the generation of random permutations is conceptually one of the easiest examples of random generation where the draw of individual items (here positions in the permutation) observes interdependencies. Random permutations are basic ingredients for random generation of other discrete structures, in particular graphs.

Generating such permutations is relatively costly. One issue that causes this high cost is the use of (pseudo-)random number generators (*PRG*), but it is not the only one: the random (!) memory read pattern of the classical shuffling algorithm, see Moses and Oakford (1963); Durstenfeld (1964) and also (Knuth, 1981, Sec. 3.4.2), implies cache misses for almost all memory accesses. Thus the performance is in general dominated by the CPU to memory latency. Neither augmenting the speed of the CPU nor the memory bandwidth would improve the performance, only augmenting the frequency of the interconnection bus would do.

In addition, for commodity architectures the performance growth is nowadays only assured by augmenting the parallelism of the CPUs, via multiple processors, cores or pipelines or via hyperthreading. In such parallel settings the random shuffle algorithm as mentioned above doesn't scale, it is inherently sequential.

In Gustedt (2007) we have shown that random shuffling of data can be realized with *linear* resource usage, CPU time as well as bandwidth, and this for a large variety of paradigms, in particular parallel, distributed and out-of-core computation. As a side effect of the controlled access to communication (resp. storage) it also shows also how the latency problem from above can be avoided.

One drawback of that algorithm is that its parallelization cost imposes a factor between 3 and 4 in the overall work: it performs two local shuffles on each parallel processor and adds one total data exchange to the work. Thus it its not suitable to lower the execution time in settings with only a few processors, such as bi- or quadri-cores.

The aim of this paper here is to show that in the case that we restrict ourselves to the common case of permutations of integers and if we only suppose PRGs as a source of indeterminism more becomes possible: a new algorithm is presented that has sublinear inter-processor communication and that has such a low parallelization cost that it performs very satisfactory on multi-core machines.

The first bottleneck that we tackle in this paper (Sec. 3) are the bandwidth requirements (as opposed to latency). A lower bound for the communication that has to be effected between different processors is given by a counting argument; the number of possible re-distribution of the elements on the target processors limits the amount of information that has to be exchanged between the processors from below. We show how the interprocessor communication can be reduced to this (sublinear) information theoretic minimum. This can still be done when assuming *full randomness*, i.e when all random decisions that we make are given by an abundant sequence of random bits.

But in the common case that we use pseudo-random numbers (instead of real randomness) the lower bound doesn't hold any more: the amount of solutions is limited by the state space of the PRG. Therefore the minimum information that has to be transferred is just that, the state space of the PRG. In Sec. 4 we show how this idea can be

used to lower the communication to only a use of bandwidth that is proportional to the used real randomness.

Sec. 5 introduces the engineering part of the present work, namely the basis of the implementation, parXXL, the explicit use of integer types of different widths, and special floating point capacities. Then it focuses on the implementation of a range coder, Sec. 5.1, and universal hash functions, Sec. 5.2, that are needed as subroutines to implement the algorithms effectively. In Sec. 6, we then report on large scale experiments that prove the efficiency and practicability of our approach in different settings: sequential execution, for parallel execution with multi-processor multi-core machines and for the distributed setting of clusters.

2 Randomized distributed shuffling and the generation of integer permutations

The distributed setting For the simplicity of the arguments we will restrict ourselves to a variant of BSP architectures (see Valiant (1990)) coined PRO (see Gebremedhin et al. (2006)): a homogeneous set of p processors, each a RAM equipped with a substantial amount of private memory and with a network device to perform point-to-point communication. For all computations the problem data of size M is supposed to be equally shared between the processor, so in particular the private memory of each has at least size $m = M/p$. To avoid complicated communication patterns and sensitivity to network latency, p is supposed to be bounded by some function in M. For technical purposes this bound usually is fixed to $\sqrt{M}$, but for our convenience at some point below we will be more restrictive, namely by imposing $p \ll \sqrt{M/\ln M}$.

Algorithms designed for this computational model easily translate into parallel or out-of-core algorithms. But for the sake of clarity we will avoid to make these explicit in the following.

2.1 Random shuffling: three different choices to make

First let us consider the *distributed shuffling problem* where we have a sequence of M items in total on p processors, each possessing a sub-sequence of $m = M/p$ items. The goal is to mix these items such that afterwards each processor again holds m items. Conceptually such a shuffling is determined by the following choices:

matrix Choose a communication matrix $A = \{a_{i,j} \mid 0 < i, j \leq p\}$, each $a_{i,j}$ holding the number of items processor i sends to processor j.

partition For any processor i partition its set of items into p subsets $P_{i,1}, \ldots, P_{i,p}$ of size $a_{i,1}, \ldots, a_{i,p}$.

local mix For any processor i shuffle its receive sets $P_{1,i}, \ldots, P_{p,i}$ to obtain a new sub-sequence.

Any given shuffling algorithm will not necessarily proceed in doing these choices as separate steps, but it will always do them, at least implicitly. The contribution in Gustedt (2007) was to show that it is in fact possible to realize each of these steps separately

and still obtain a uniformly distributed random permutation of the items with a resource usage that is linear and equally shared between the processors. Evidently the most difficult part for that is the choice of the matrix A which was shown to be distributed with some generalization of a multivariate hypergeometric distribution, and for which efficient (sequential, parallel, ...) sampling algorithms were given.

Procedure `ParIntPerm` gives a of the algorithm that on each processor first generates a source table V', permutes it, and sends it out in pieces to all other processor according to a communication matrix A. For the sake of readability the algorithm is simplified in that it supposes that the share m of elements is equal for all processors, for sending as well as for reception.

Procedure `ParIntPerm`(m, p, ν) : Parallel Random Integer Permutation

Input: Non-negative integers m (local size), p (amount of processors) and ν the id of the processor

Output: Table $V = V[1], \ldots, V[m]$ such that the sets of all $V[i]$ on all processors represent a permutation of the integers $1, \ldots, p \cdot m$.

matrix All processors collectively choose $A = (a_{i,j})$, the communication matrix

partition **begin**

  Create a table V' with $V'[1] = (\nu - 1) \cdot m + 1$, ..., $V'[m] = \nu \cdot m$

  Permute V'

  starting with $V'[1]$, **for** $j = 1, \ldots, p$ **do** set B'_j to the next block of $a_{\nu,j}$ elements in V'

  starting with $V[1]$, **for** $j = 1, \ldots, p$ **do** set B_j to the next block of $a_{j,\nu}$ elements in V

end

exchange **begin**

  for $j = 1, \ldots, p$ **do** send B'_j to processor j

  for $j = 1, \ldots, p$ **do** receive B_j from processor j

end

local mix Permute V

2.2 The performance bottleneck: communication volume

In addition to the choices introduced above a shuffling algorithm also has to exchange the data. In `ParIntPerm` this is done in the block labeled exchange. If the contents of the data items is not deducible from other sources, the minimum communication volume that has to be exchange is essentially the entropy of that data. The only way to save here are cases where part of the data stays on the same processors, but obviously the size of the resident data on any processor is 0 for the worst case and $\Theta(m/p) = \Theta(M/p^2)$ on expectation.

With these bounds in mind the overall approach as described is asymptotically optimal for any coarse grained setting such that p the processor number is small compared to the size of the data, more precisely if $p \in O(\sqrt{M})$, see Gustedt (2007). In particular it is dominated by the communication that is to be performed if the data is not compressible.

But for the case the goal is only to provide a random permutation of integers, this lower bound does not necessarily hold anymore. There a naive and direct exchange of the items would require $\Omega(M)$ machine words in total (or $\Omega(M \log M)$ bits) to be communicated. We will see in the next section (Sec. 3) that the entropy then in fact is smaller than that compression with a compression ration of $\log p / \log M$ can be used to reduce the communication to a sublinear number of machine words.

2.3 The pitfalls of pseudo-randomness

All the investigations done so far on the subject (implicitly or explicitly) supposed that an unbiased source of randomness is available to the algorithm, which is an assumption that is relatively unrealistic. Some real randomness is in fact available on modern platforms (e.g Linux' `/dev/random`), but generally the access is slow and costly. So pseudo-random generators (PRG) are chosen as a partial replacement for real randomness. But then a pigeon hole argument shows that the permutation space from which we draw is drastically limited: if the state space of the PRG on each processor is B bits, say, the maximal amount of different permutations that might be produced is 2^{Bp} whereas the total number of permutations is $M! \approx e^{M \ln M} \in 2^{O(M \log M)}$. So unless B can be chosen to $\Omega((M \log M)/p) = \Omega(m \log M)$ most of the valid permutations will even be left out by such an algorithm.

This restriction applies particularly for any implementation of PRG with a state space of fixed size as they are commonly found on today's platforms.

A common technique to circumvent this inherent design difficulty is to reinitialize the PRGs from time to time with real random bits from sources as described above. Usually PRGs are implemented using large cycles of an (or several) arithmetic progression(s). Initializing several PRG using the same cycle space introduces subtle interdependencies between them. Thus the quality of the pseudo-random generation can not be guaranteed under such circumstances, so even less the quality of derived 'randomly' chosen objects like permutations.

By the same argument follows that already the parallel design of using the same PRG with different initialization of the statespace is suboptimal. Initializing a PRG on each processor using the same cycle space introduces the same subtle interdependencies between them as would a reinitialization in a sequential setting.

In Sec. 4 we will propose a setting that allows to chose the amount of real randomness B that is invested into the permutation generation. Part of that randomness is then used to sample the communication matrix and the other to initialize the statespace of an independent set of PRGs. These then are used to generate the permutation in place on the processors and the communication volume is dominated by exchanging it between the processors.

By that we reduce the communication asymptotically to the entropy of the solution space, the at most 2^B permutations that are reachable with the chosen amount of randomness. Since this approach uses different PRGs for all processors it also avoids the interdependency problem as mentioned above.

3 Reducing communication under full randomness assumptions

So any shuffling algorithm that randomly distributes M data between p processors has to perform a partitioning of the data into p equally sized buckets, namely the subsets of elements that land on the same target processor. Such a partitioning of M items on p processors is equivalent to assigning numbers $1, \ldots, p$ to the items. So we may easily encode it with $O(\log p)$ bits per item. Some straight forward computation shows that at least $\Omega(M \log p)$ bits are needed to encode a partition into p blocks of equal size. So the total coding size is $\Theta(M \log p)$.

The goal of this section is to show that this amount of information ($\Theta(M \log p)$) is not only sufficient to describe such a partition but that in our case the corresponding communication can be effected within that bound.

Proposition 1 *A partition of the integers $1, \ldots, M$ into p parts of size $m = M/p$ on p processors can be communicated within a send and receive size for each processor of $O(p \log M + m \cdot \log p)$ bits and with a linear computational overhead.*

The key to communication reduction is a specific compression technique that is well adapted to our context. Using different compression algorithms for communication (in a broad sense) has already been described in various previous work, see e.g Curewitz et al. (1993); Jeannot et al. (2002); Davis et al. (1998); Bordawekar et al. (1996). Generally the application framework there is relatively wide such that not much about the entropy of the information that is communicated is known. Thus usually the compression gain can only be studied empirically.

As seen above, our case here is different in that we know the amount of information that we have to communicate. In fact, we may already assume that we have an efficient encoding of our partition by $\lceil \log p \rceil$ bits per element. If on each processor the local partition (of size $m = M/p$) is given as lists or arrays of the different parts, we can easily produce a table $T[1], \ldots, T[m]$ of m entries consisting each of $\lceil \log p \rceil$ bits.

To be able to communicate efficiently we have to separate out the 'bits' that are to be send to each individual other processor Q. We do that by taking all elements that go to Q in ascending order (using table T) and by encoding this sequence by the difference between successive elements. Procedure `CompressPartition` summarizes such a procedure that does this encoding 'on the fly' for all target processors.

Each individual difference d that we compute in **differ** can be large. But if we look at the total sequence of such differences that a target processor will receive from all others we see that their average is $M/m = p$.

In **cram** we encode a segment between two occurring elements with an alphabet of two symbols ('0' and '1'), namely by inserting d '0's followed by a '1'. Because we know that '1's only occur with a probability of $1/p$ we can use range encoding, see e.g Martin (1979), to encode the overall sequence for any target processor Q with $O(m \cdot \log p)$ bits. It is straight forward to see that all this can be produced and communicated with an overhead of at most $O(p \log M)$ bits per processor so we obtain an overall amount of $O(p \log M + m \cdot \log p)$ bits that each processor receives.

Procedure `CompressPartition`(o, m, p, a, P)

Input: Non-negative integers o (start offset), m (local size) and p (amount of processor)
$a = (a[1], \ldots, a[p])$ with $m = \sum_i a[i]$, the row of the communication matrix
$P = (P[1], \ldots, P[m])$ with $\{x \mid x = P[i] \text{ for some } i\} = \{y \mid o < y \leq o + m\}$;
Output: compressed streams $(C[1], \ldots, C[p])$, $C[i]$ representing a part of P of size $a[i]$.
Use o to compute $T = (T[1], \ldots, T[m])$ such that $T[i]$ is the target processor for $P[i]$
Initialize $C = C[1], \ldots, C[p]$ to all empty
Initialize $V = V[1], \ldots, V[p]$ to all 0
foreach $i = 1, \ldots, m$ **do**
  Set $t = T[i]$ the target processor of element i
differ Set $d = i - V[t]$, the difference of i to the previous element for processor t
cram Append d '0's and a '1' to $C[t]$
  Set $V[t] = i$

A symmetric argument shows that no source processor has to send out more than $O(p \log M + m \cdot \log p)$ bits. So the communication scheme remains balanced between the processors.

For the claimed linearity of the algorithm we have to adapt the range coding in **cram** such that it encodes several '0's at once, resulting in amortized constant time per execution of **cram**. The details of this will be presented in Sec. 5.1.

4 Linking the use of random bits to bandwidth and quality

Procedure `GenPerm` partially presents a new algorithm that replaces large parts of the communication in `ParIntPerm`. The main idea is that instead of communicating an already permuted integer table, this first permutation is "*emulated*" directly on each of the target processors. Supposing that processor i would have used permutation π_i to permute its data, it communicates its inverse $\mu_i = \pi_i^{-1}$ such that each target processor is able to compute the elements that it would have received from i. Then, as before a locally computed permutation is used to write the generated elements in random order.

`GenPerm` uses universal hash functions as a tool for the local permutations. In the domain of integers universal hash function are simply bijections for which we have an effective algorithm to evaluate them for each item. This possibility of a local evaluation is crucial for our approach, since we will nowhere compute the permutation of a source part at once, but only compute it in p^2 pieces, on each processor p pieces for all p different processors.

Universal hash functions have been much discussed in the context of cryptography, see e.g Nevelsteen and Preneel (1999). `GenPerm` is generic in that it does not specify

Procedure `GenPerm`(m, p, ν, a, S, P) Generate a random integer permutation in place

Input: Non-negative integers m (local size), p (amount of processors) and ν the id of the processor
$a[1], \ldots, a[p]$ with $m = \sum_i a[i]$, the column of the communication matrix
$O[1], \ldots, O[p]$ the offsets of the source buckets in the local table on processor i
$U[1], \ldots, U[p]$ states of universal hash functions μ_i on $1, \ldots, m$
Output: $V = V[1], \ldots, V[m]$, the local part of the target permutation.
$t = 0$
Generate a new universal hash function γ
foreach $i = 1, \ldots, p$ **do**
  Set $o = (m(i-1)) + 1$ the offset of the table of processor i
  Initialize μ_i from $U[i]$
  foreach $k = O[i], \ldots, O[i] + a[i] - 1$ **do**
preimage Set $k^{-1} = \mu_i(k)$, the pre-image of k under $\pi_i = \mu_i^{-1}$
generate Set $K = o + \mu_i(k^{-1})$, the element that would have been sent
permute Set $t' = \gamma(t)$, the final position of K
store $V[t'] = K$
    $t = t + 1$

what hash function is to be used. The amount of randomness that such a function brings will directly depend of the choice of the family, its size and the size of the state data that is to be communicated to make an efficient computation independently possible on all processors.

Besides the use of the hash functions the other source of randomness in `GenPerm` is given by the communication matrix. The more processors, the larger is the matrix and the more randomness it may bring into the solution. If we want to tune the amount of randomness that is used for the solution the dependency from an architectural parameter such as p is not desirable. `GenPermBlock` avoids this by dividing the problem into more blocks, b per processor.

Procedure `GenPermBlock`(m, p, b, ν, a, S, P) Generate a random integer permutation in place

Input: Non-negative integers m (local size), p (amount of processors), b (blocks per processor) and ν the id of the processor
`/* a, S, P as in ParIntPerm, only indices are 1,...,p·b */`
Output: $V = V[1], \ldots, V[m]$, the local part of the target permutation.
`/* same as ParIntPerm */`
foreach $i' = 1, \ldots, p$ **do**
  foreach $i'' = 1, \ldots, b$ **do**
    $i = (i' - 1) \cdot b + i''$
    `/* same as ParIntPerm */`

`GenPermBlock` now gives us two parameters that control the amount of randomness: (1) b the number of blocks per processor and (2) the family of universal hash functions. They both have and impact on the randomness *and* the performance, and both can be chosen in some way to obtain a complete random permutation.

If $b = m$, *i.e* each block holds exactly one element, A simply becomes a matrix of 0 and 1, each element $a_{i,j} = 1$ indicating the final position j of element i. Thus A then is a permutation matrix. These sparse matrices allow for the same kind of encoding tricks as were presented above. So effectively, the two extremal cases, no division in blocks but the whole randomness in the local shuffle, and complete subdivision without local shuffle, amount to the same.

5 Engineering

The implementation of the algorithms was undertaken with `parXXL`[1], a C++-library that allows experimenting and benchmarking of unmodified code on different types of architectures, parallel machines or clusters.

To give an idea of what we are heading for, let us look at the performance of the sequential shuffling algorithm that is implemented in parXXL. This implementation is already quite efficient, since it uses some prefetching techniques to circumvent the latency problems that were mentioned in the introduction.

That implementation needs about 140 *ns* per item for a 64bit integer permutation on a 1.8 *GHz* PC (i86_64 architecture). This corresponds to roughly 250 cycles. These numbers are basically against what we have to compete with an alternative implementation and which should also enable us to judge the parallel efficiency: the time processor product per item should not exceed these 140 *ns* by much.

We also will have to take the time for the sampling of the communication matrix A into account. The computing time for that is dominated by draws of a hypergeometric distribution which takes about 1 μs in the same setting, based on the standardized PRG `jrand48`[2]. Since the size of that matrix grows quadratic in the number of buckets in which we split the problem, we will have to be careful not to subdivide the problem too much. The implementations that are described here are based on the matrix generation that is already found in parXXL. Unfortunately it is not yet completely parallelized, which we will see to be an issue for the benchmarks, see Sec. 6.

Since we will implement algorithms that go down to the bit level of the represented data another issue that has to be handled carefully is the wordsize of the target architecture. Even talking about "*the*" wordsize is generally not possible. Modern hybrid architectures may use different constants for different types of addressing, e.g 36 bits for physical addressing, 48 bits for virtual addressing, and 64 bits to represent pointers. Arithmetic can be performed with varying efficiency if the data are 32 bit integers (`uint32_t`), 64 bit integers (`uint64_t`), floating point numbers (`double`), or of some platform specific register vectors, such as the `i386`'s `SSE` registers.

To be able to realistically represent large integer permutations we will assume that the *final* output will be a table of m `uint64_t`. But arithmetic on this type may be

[1] http://parxxl.gforge.inria.fr/

[2] http://opengroup.org/onlinepubs/007908799/xsh/drand48.html

slow (in particular division and modulo) and storage (and bandwidth) might be wasted if we represent small numbers with it. For the implementation we therefore distinguish the target datatype from intermediate ones that are used during the computation, in particular `double` for range encoding and `uint32_t` for universal hash functions. We provide a generic C++ `templates` implementation that depends on two *type* parameters, one for the target type and one for the intermediate type. This enabled us to chose them easily in function of the target architecture.

5.1 Range encoding

Range encoding (see Martin (1979)) is a particular case of entropy encoding that is asymptotically optimal. That is, it encodes a string over an alphabet Σ according to the probability $P(\sigma)$ of the occurrence of the individual symbols $\sigma \in \Sigma$. Under the assumption of independence of the occurrence of the symbols, the length of the encoding tends towards the information theoretic optimum.

It views the encoded string (the code) as a big binary number C. Its name comes from the fact that during the encoding phase it works with a lower and upper bound C^- and C^+ that define a *range* within the final code will be found. Each occurrence of a new symbol $\sigma \in \Sigma$ restricts the actual range to a new range with a size that is proportional to $P(\sigma)$.

The particularity in our context for the range encoding needed for `Compress-Partition` is that we need to encode long runs of '0's efficiently. A commonly used trick to cope with that is to add artificial symbols to Σ that represent long runs. Whereas such an approach is fast on the coding side, it requires a binary search for the encoded artificial symbol on the decoding side. Thus it has some overhead that is proportional to the logarithm of the length of the run.

To cope with that we use `doubles` to represent the 'interesting' part of the bounds, i.e that part of the bounds that are yet subject to change during encoding or decoding. IEEE[3] `doubles` are normalized to have 52 bits in the mantissa, from which we use 48 for our implementation. They have the advantage that their order of magnitude is automatically maintained in the exponent and that is accessible through cheap bit operations. By that an estimation of the length of the next run can easily be obtained by an integer logarithm operation, on the decoding site.

5.2 Families of universal hash functions

Since the goal of our implementation is first of all to show the potential of the approach we chose some relatively simply universal hash functions:

- The universal hash functions must be fast.
- They must be independent for all processors.
- They must allow for a controlled trade-off between their state-size and their efficiency.

[3] http://

A simple well-known such family is given by an arithmetic progression:

$$\Pi^{\rho}_{\alpha,\beta}(x) := \alpha \cdot x + \beta \pmod{\rho} \tag{1}$$

Where ρ is a prime number and $0 < \alpha < \rho$, $0 \leq \beta < \rho$ are some fixed parameters. Since by definition α and ρ are mutually prime, it is easy to see that for any such choices $\Pi^{\rho}_{\alpha,\beta}$ is a permutation on $\{0, \ldots, \rho - 1\}$. In addition, if we fix ρ, the choice of α and β gives us $\rho \cdot (\rho - 1)$ distinct functions $\Pi^{\rho}_{\alpha,\beta}$: for two distinct choices of β the images of $x = 0$ are distinct, and for two distinct choices of α_1 and α_2 the image $y = \alpha_1\alpha_2 + \beta \pmod{\rho}$ has different pre-images, namely α_2 and α_1.

We need universal hash functions that operate on any interval $[0, \ldots, m - 1]$, not only for prime numbers. Procedure ${}^{1}\texttt{uhash}^{\rho,m}_{\alpha,\beta}$ generalizes the family $\Pi^{\rho}_{\alpha,\beta}$ to general m by simply following a cycle of the permutation $\Pi^{\rho}_{\alpha,\beta}$ that might lead outside of the range $[0, \ldots, m - 1]$ until it leads back into it. Again, it is easy to see that ${}^{1}\texttt{uhash}^{\rho,m}_{\alpha,\beta}$ defines a permutation and that for fixed m and ρ these permutations are all pairwise distinct.

Procedure ${}^{1}\texttt{uhash}^{\rho,m}_{\alpha,\beta}(x)$ universal hash function with twist 1.

Input: Non-negative integers x (input), ρ (prime), m (domain), such that $x < m \leq \rho$
α (factor) and β (additive shift), with $0 < \alpha < m$ and $\beta < m$
Output: Non-negative integer $y < m$, such that for all $x_1 \neq x_2 < m$, ${}^{1}\texttt{uhash}^{\rho,m}_{\alpha,\beta}(x_1) \neq {}^{1}\texttt{uhash}^{\rho,m}_{\alpha,\beta}(x_2)$
repeat $x = \Pi^{\rho}_{\alpha,\beta}(x)$ **until** $x < m$
return x

Procedure ${}^{t}\texttt{uhash}^{\bar{\rho},m}_{\bar{\alpha},\bar{\beta}}$ shows the generalization of ${}^{1}\texttt{uhash}^{\rho,m}_{\alpha,\beta}$ by concatenating ${}^{1}\texttt{uhash}^{\rho,m}_{\alpha,\beta}$ t times.

Procedure ${}^{t}\texttt{uhash}^{\bar{\rho},m}_{\bar{\alpha},\bar{\beta}}(x)$ universal hash function with twist t.

Input: Non-negative integers x (input), t (twist), m (domain), $\bar{\rho} = (\rho_1, \ldots, \rho_t)$ (prime numbers), $\bar{\alpha} = (\alpha_1, \ldots, \alpha_t)$ (factors) and $\bar{\beta} = (\beta_1, \ldots, \beta_t)$ (additive shifts), such that for all i: $x, \beta_i < m \leq \rho_i$ and $0 < \alpha_i < m$
Output: Non-negative integer $y < m$, such that for all $x_1 \neq x_2 < m$, ${}^{t}\texttt{uhash}^{\bar{\rho},m}_{\bar{\alpha},\bar{\beta}}(x_1) \neq {}^{t}\texttt{uhash}^{\bar{\rho},m}_{\bar{\alpha},\bar{\beta}}(x_2)$
return ${}^{1}\texttt{uhash}^{\rho_1,m}_{\alpha_1,\beta_1}\left({}^{1}\texttt{uhash}^{\rho_2,m}_{\alpha_2,\beta_2}\left(\cdots {}^{1}\texttt{uhash}^{\rho_t,m}_{\alpha_t,\beta_t}(x)\cdots\right)\right)$

In our implementation we chose ρ, the prime number, deterministically based on m and on the ID of the processor. This ensures that all these prime numbers are different for all processors, and that all processors may compute them without the need to exchange them. Only the constants α and β are chosen randomly for each processor and are then exchanged.

platform	compiler version	nodes	per node processor	cores	speed	cache	RAM
damogran	gcc 4.1.3	1	intel x86_64	2	1.80 GHz	2 MiB	3.86 GiB
grelon	gcc 4.1	120	intel x86_64	4	1.60 GHz	4 MiB	1.97 GiB

Table 1: Platform summary

Choosing p prime numbers can be done efficiently if we restrict ourselves to the case where $p \ll m/\ln m$. We just test the values of $m, m+1, \ldots$ for primality. Because of the known density of prime numbers of about $1/\ln x$ we are sure to find enough prime numbers in the range $[m, 2m)$ with an amortized computational overhead that does not exceed $O(m)$ on all processors.

The computational cost of $^1\texttt{uhash}^{\rho,m}_{\alpha,\beta}$ in our context is dominated by the number of evaluations of $\Pi^{\rho}_{\alpha,\beta}$. Since as a whole we could have to run through all cycles of the permutation this number may be ρ. So, $^1\texttt{uhash}^{\rho,m}_{\alpha,\beta}$ could be very expensive in our setting, if the range m and the prime number ρ were of different orders of magnitudes. But fortunately, as seen above, we may restrict ourselves to the case that $\rho < 2m$ and thus the total number of calls to $^1\texttt{uhash}^{\rho,m}_{\alpha,\beta}$ *per source processor*[4] is $O(m)$.

Another important issue to obtain a competitive implementation of $\Pi^{\rho}_{\alpha,\beta}$. Here the non-trivial operation is taking the modulus. CPUs differ greatly on the efficiency of that operation not only between different CPUs but also on the same CPU for data types of different width. In particular on the target platforms, all equipped with `i86` processors, modulus for 32bit integers is quite efficiently done by a single instruction in some clock cycles. For 64bit integers this might be synthesized in software and take much longer. Therefore it was crucial for the success of the implementation to eventually split the problem in more than p subranges, as was presented in `GenPermBlock`. Hereby we ensure that the local indices for each block do not exceed 32 bit, i.e that the blocks have less than 2^{32} elements.

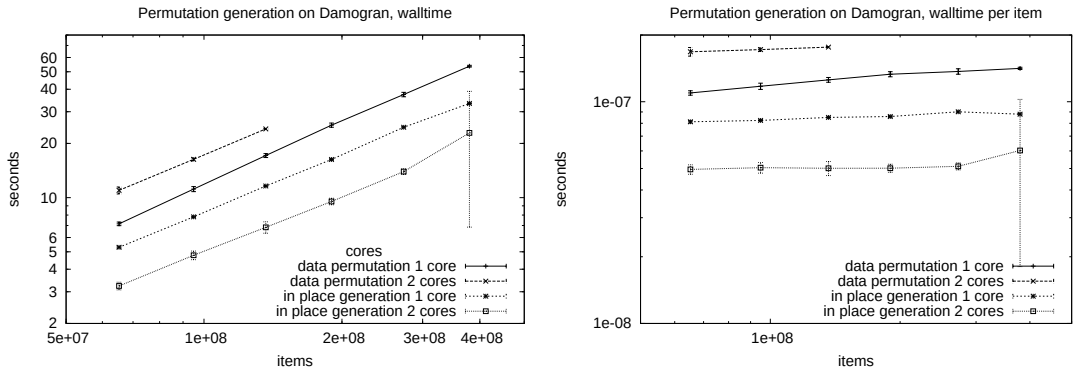
6 Experiments

We will present experiments on two different platforms, one a laptop computer "*damogran*" and the other a compute cluster "*grelon*", part of Grid5000[5]. The algorithms that were implemented is a variant of the classical shuffling algorithm, the parallel data permutation algorithm of Gustedt (2007), and the in place generation algorithm of this paper with different strategies for the block sizes but with fixed hash strategy $^1\texttt{uhash}^{\rho,m}_{\alpha,\beta}$.

The programs were benched in a "reasonable" range of problem sizes: the maximal value M^+ is generally the size that still fits into the platform's RAM. From there other smaller values corresponding to $M^+/2^{i/2}$ for some values of i were also tested. Each data point in the graphs corresponds to the average over 20 runs. In addition, some

[4]This amortization only holds per source processor, an individual target processor could be overcharged when he would have to run through a lot of cycles. It is possible to avoid such a potential imbalance by computing and communicating these cycles of the permutations in advance. We will see below this was not relevant for the experiments, so such a strategy was not implemented.

[5]http://www.grid5000.fr



(a) total time (with error bars)

(b) time per item (with error bars)

Figure 1: Run time comparison on bi-core

figures show error bars for the computed variance of the results, but in most cases the variance is so small that this is not noticeable.

To emphasize on the scaling properties and proportions, the results are represented in doubly-logarithmic scale. Data points are chosen such that every second point roughly corresponds to a doubling in size (or processors), *i.e* each step is about $\sqrt{2}$ from the previous.

Fig. 1 shows a comparison of the different permutation programs on *damogran*. We see that shuffling takes about 110 to 140 *ns* per item. The parallel data permutation algorithm for two processors slows down to about 170 *ns*. In fact the break even point for this parallel algorithm lays between 3 and 4 processors, so the parallelization for this restricted parallelism of only two cores is not yet worth it, see also Gustedt (2007).

Compared to that, the new generation algorithm with 310 to 590 blocks already shows a speedup when only executed on 1 core (80 to 90 ns) and improves to 50 ns when run on 2 cores. Fig. 2 plots the speedup and slowdown values for the possible comparisons. Observe also that already for the smallest value of 2^{26} items the total amount of permutations in the sample space is about $e^{26 \cdot 3.258} \approx e^{84.7} \approx 2^{122.2}$. So a pseudo-random generator with a state of at least 123 bits would be required to cover the whole sample space. The `rand48` routines that are used for the implementation have the advantage that they are quite fast but only hold a state of 48 bits. Without additional cost, our in place generation here would be able to take advantage of some thousand real random bits (a hash function state per block) that were obtained from `/dev/random`.

For completeness, Fig. 3 shows the computing time of an entropy encoding. Here the measurement is quite involved since we first have to benchmark the encoding algorithm together with the random process that generates the data, then we have to benchmark the process without the encoding and the difference is then taken as the time for encoding. As we see the sum of encoding and decoding is between 180 and

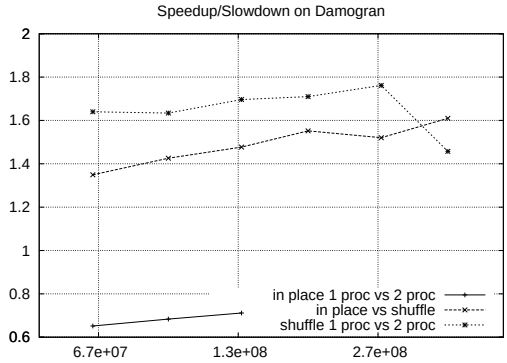


Figure 2: Speedup or slowdown on bi-core

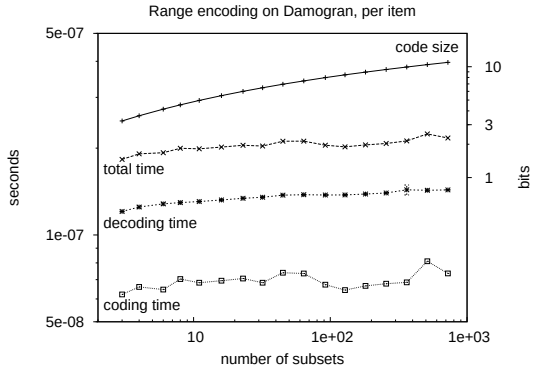


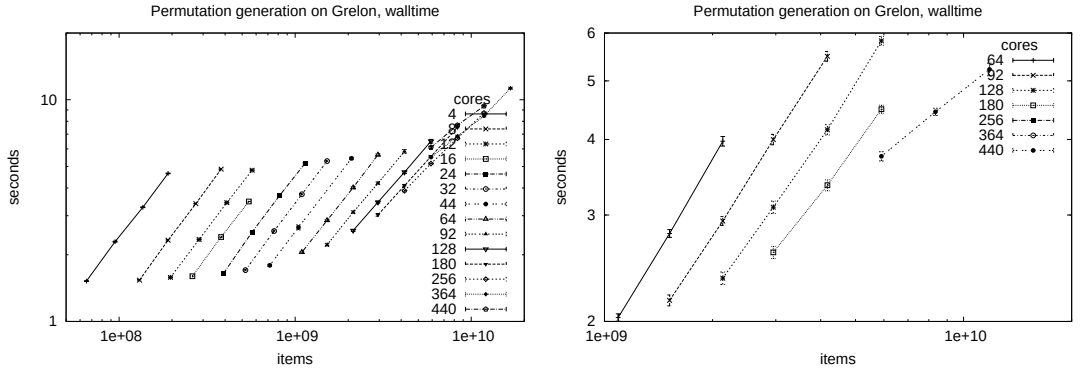
Figure 3: Compression by range encoding, sequential

200 *ns*, much slower than the random shuffling itself. This corresponds to a throughput of 35 to 40 *MB/s* on the network link, too restrictive in most of today's computing environments to pay off. Therefore we did not push the implementation of that setting further.

The cluster experiments on "*grelon*" follow two different strategies to determine the number b of blocks per processor of `GenPermBlock`. The first strategy uses a heuristic value of about $\frac{\sqrt{M}}{\log_2 M}$ that is meant to warrant that the computation of matrix doesn't dominate the problem. The other strategy is to fix b to 1024.

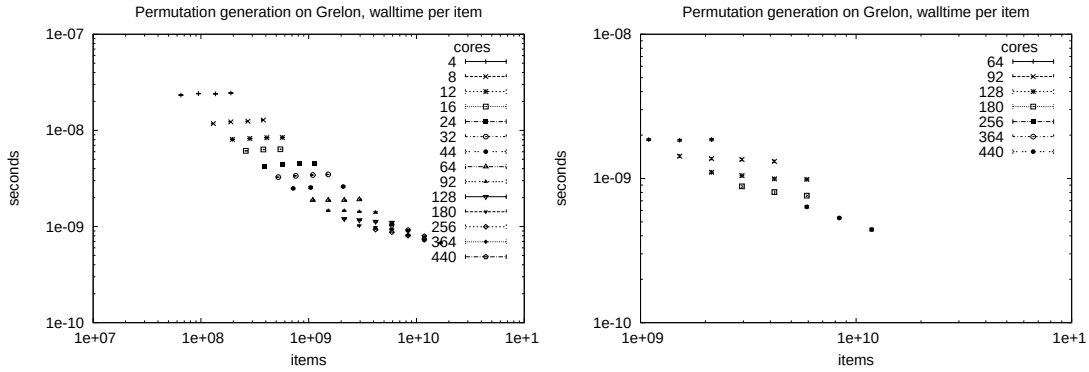
Fig. 4 gives the average running times for experiments within two orders of magnitude for the problem size and for the number of processors. The plots show very good scaling of the programs, the progression of the time with the data size are straight lines and the error ranges are invisible.

As the problem sizes concern different orders of magnitude a direct comparison by means of "speedup" plots as given above is not possible. Instead, Fig. 5 shows the same data as before but now the computing times are given as *seconds per data item*.



(a) $\approx \frac{\sqrt{M}}{\log_2 M}$ blocks, variable (b) 1024 blocks, fixed

Figure 4: Cluster experiments with two different strategies for the block sizes, total times with error bars

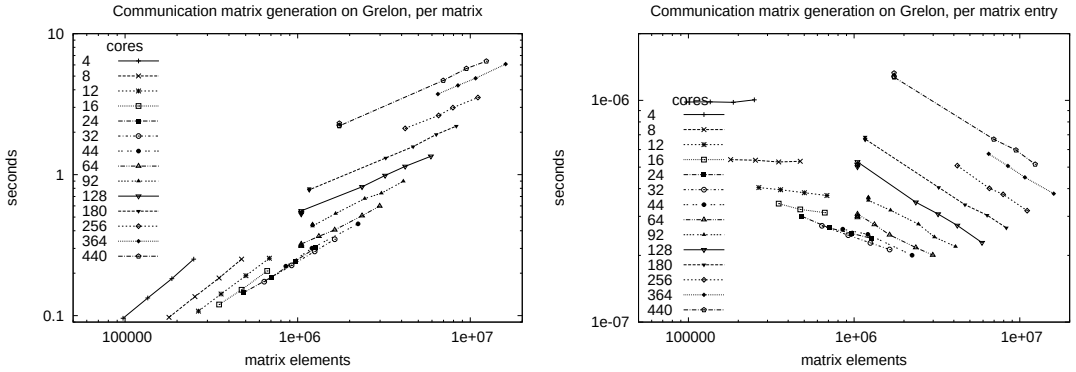


(a) $\approx \frac{\sqrt{M}}{\log_2 M}$ blocks, variable (b) 1024 blocks, fixed

Figure 5: Cluster experiments with two different strategies for the block sizes, amortized times per item

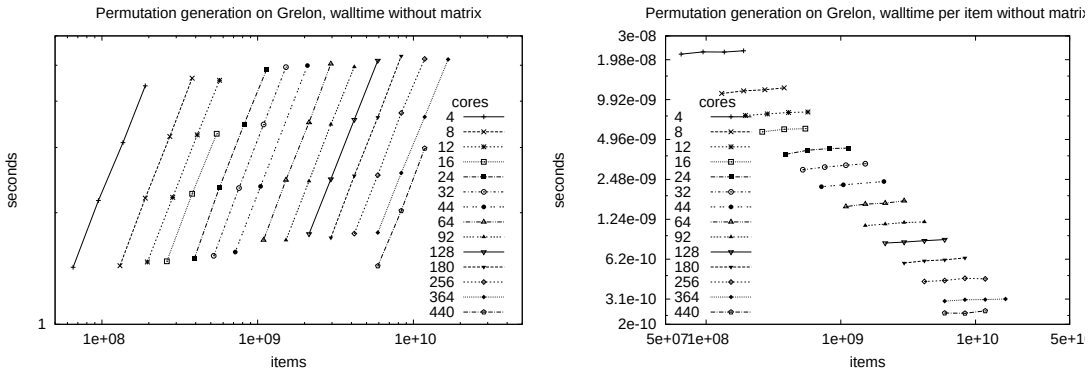
The plots are mainly horizontal lines, meaning that in fact the total running times are linear in the number of items. This holds up to 440 processor cores. To see why for this large amount of processors the pattern is lost, we give the running times of the matrix generation (Fig. 6). We see that the running time for the matrix generation that were used with our heuristic only improves up to 32 processors where it stalls, and then increases where it uses more processors for larger matrices.

The in place generation itself (figures 7 and 8) without the matrix generation is again very regular.



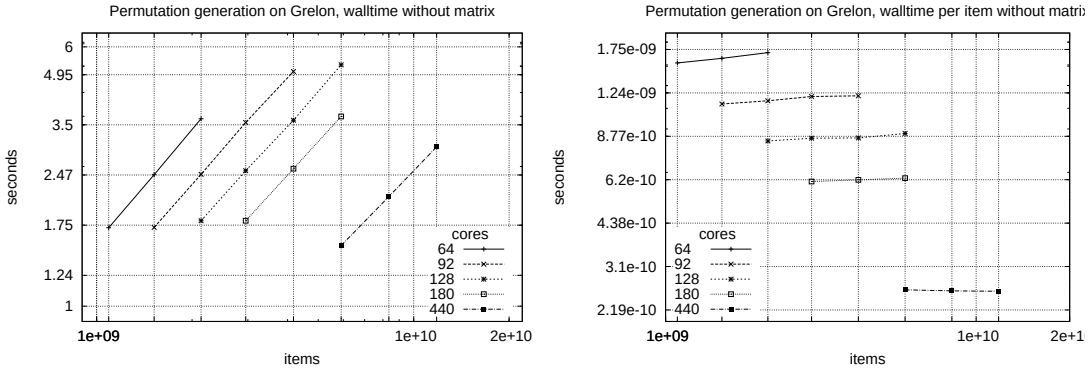
(a) time per matrix (b) time per matrix entry

Figure 6: Cluster experiments, matrix generation



(a) total time (b) time per item

Figure 7: Cluster experiments with variable block size, times without matrix generation



(a) total time

(b) time per item

Figure 8: Cluster experiments with fixed block size 1024, times without matrix generation

7 Conclusion and Outlook

With the present work we show that for the generation of integer permutations there are alternatives to the classical data shuffling algorithm and to other more sophisticated redistribution algorithms. The shuffling algorithm doesn't scale in the first place; it is inherently sequential and is not suited to a computing world that consists of distributed multicore machines. Redistribution of (generated) data on the other hand doesn't use the information-theoretical potential. Both share the problem that they generally use PRGs to draw random positions of items. The state space of these PRGs is easily underspecified and does not allow to cover the whole sample space.

Our first approach of using range encoding to limit communication to the information-theoretic bound shows to be as compute consuming as the shuffling algorithm itself. So it can only be competitive in a restrictive setting where bandwidth is very limited compared to computing power, probably less then 10 *MB/s* for today's platforms.

The second approach avoids the initial (and artificial) generation of the identity permutation and generates the target permutation in place. The only data that is communicated between the processors are parts of a communication matrix and state vectors of universal hash functions. This allows for a fine grained control and trade off between the investment of random bits for the solution and the quality and running time.

This second approach improves over the previously know ones, in sequential and in parallel. The parallel implementation shows real and effective speedups and sizeups, for clusters and multi-cores as they become more and more dominant today.

The experiments also showed some problem of the current implementation, namely the generation of the communication matrix. Here, in a future work the parallelization will be driven further to be able to tackle multi-cluster environments of perhaps several thousands of nodes or cores.

Another limitation that showed up during this work is the lack of accepted quality measures for random permutations. What would be a good statistical test that a fam-

ily of generated random permutations would have to pass? The lack of such a quality measure also made it pointless for the time being to try other hash functions, such as ${}^{t}\mathtt{uhash}^{\bar{\rho},m}_{\bar{\alpha},\bar{\beta}}$. An interesting future study could be to compare the gains of randomness that are obtained by more complicated hash functions and/or by augmenting the amount of blocks into which the problem is subdivided.

A more prospective use of the present work is to construct other random structures, such as graphs and also for new PRGs from these newly available permutations. The general idea of our approach allows for a controlled way to obtain randomness that is completely independent on different processors. It uses *some* real randomness to initialize the state in of an iteration process that is different on each processor.

References

Bordawekar, R., Choudhary, A., Ramanujam, J., 1996. Automatic optimization of communication in compiling out-of-core stencil codes. In: ICS '96: Proceedings of the 10th international conference on Supercomputing. ACM Press, New York, NY, USA, pp. 366–373.

Curewitz, K. M., Krishnan, P., Vitter, J. S., May 1993. Practical prefetching via data compression. In: Proceedings of the 1993 SIGMOD.

Davis, G., Lau, L., Young, R., Duncalfe, F., Brebber, L., 1998. Parallel run-length encoding (RLE) compression reducing I/O in dynamic environmental simulations. The International Journal of High Performance Computing Applications 12 (4).

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

Gebremedhin, A. H., Essaïdi, M., Guérin Lassous, I., Gustedt, J., Telle, J. A., 2006. PRO: A model for the design and analysis of efficient and scalable parallel algorithms. Nordic Journal of Computing 14.

Gustedt, J., 2007. Efficient Sampling of Random Permutations. Journal of Discrete AlgorithmsTo be published as doi:10.1016/j.jda.2006.11.002.
URL http://hal.inria.fr/inria-00000900/en/

Jeannot, E., Knutsson, B., Björkman, M., Jul. 2002. Adaptive Online Data Compression. In: IEEE High Performance Distributed Computing (HPDC'11). Edinburgh, Scotland, pp. 379–388.

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

Martin, G. N. N., March 1979. Range encoding: an algorithm for removing redundancy from a digitised message. Presented at the Video & Data Recording Conference, in Southampton, UK.
URL http://www.compressconsult.com/rangecoder/rngcod.pdf.gz

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

Nevelsteen, W., Preneel, B., 1999. Software performance of universal hash functions. In: Stern, J. (Ed.), Advances in Cryptology - EUROCRYPT 1999. Vol. 1592 of Lecture Notes in Computer Science. Springer-Verlag, Prague,CZ, pp. 24–41.

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

Wu, K., Otoo, E. J., Shoshani, A., April 2004. An efficient compression scheme for bitmap indices. Tech. Rep. LBNL-49626, Lawrence Berkeley National Laboratory, http://repositories.cdlib.org/lbnl/LBNL-49626.

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 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