Efficient Parallel Random Sampling—Vectorized, Cache-Efficient, and Online

PETER SANDERS, SEBASTIAN LAMM, LORENZ HÜBSCHLE-SCHNEIDER, EMANUEL SCHRADE, and CARSTEN DACHSBACHER, Karlsruhe Institute of Technology

We consider the problem of sampling n numbers from the range $\{1, \ldots, N\}$ without replacement on modern architectures. The main result is a simple divide-and-conquer scheme that makes sequential algorithms more cache efficient and leads to a parallel algorithm running in expected time $O(n/p + \log p)$ on p processors, i.e., scales to massively parallel machines even for moderate values of n. The amount of communication between the processors is very small (at most $O(\log p)$) and independent of the sample size. We also discuss modifications needed for load balancing, online sampling, sampling with replacement, Bernoulli sampling, and vectorization on SIMD units or GPUs.

CCS Concepts: • **Mathematics of computing → Probabilistic algorithms**; *Random graphs*; *Random number generation*; • **Theory of computation → Massively parallel algorithms**; **Generating random combinatorial structures**;

Additional Key Words and Phrases: Hypergeometric random deviates, parallel algorithms, communication efficient algorithms

1 INTRODUCTION

Random sampling is a fundamental ingredient in many algorithms, e.g., for data analysis. With the advent of ever larger data sets ("Big Data"), the number of elements sampled from and even the sample itself can become huge. Often the subsequent processing of the sample is comparatively fast, and thus taking the sample can become a performance bottleneck. Moreover, the speed of a single processor is stagnating so that parallel algorithms are required for efficient sampling. Furthermore, we can observe that only *local* processing yields fast parallel algorithms and promises to scale linearly with the number of processors p. Processor coordination over global memory or even communication over the network quickly becomes a bottleneck [21]. This is particularly true for big data problems that often run on cloud resources with limited communication capabilities or for high-performance computing where the largest configurations are limited by the bisection bandwidth of the network.

In this article, we focus on the classical problem of sampling n numbers out of the range $1..N$ without replacement.[1] In Section 2, we discuss building blocks and previous approaches. Section 3 introduces our divide-and-conquer algorithm for sampling without replacement. We discuss a number of generalizations in Section 4, including online sampling in sorted order, load balancing, uneven distribution of the sampled universe, using true randomness, achieving deterministic results, sampling with replacement, Bernoulli sampling, and vectorization. After providing details of our implementation in Section 5, Section 6 describes experiments that demonstrate the speed and scalability of our algorithm on both CPUs and GPUs. Section 7 summarizes the results and discusses some applications.

2 PRELIMINARIES AND RELATED WORK

Our goal is to efficiently take a sample of size n from the range $1..N$ using p processors. This algorithm can also be used to sample from an array of N elements. More generally, the result applies to sampling from a set M of elements if $N = |M|$ is known and if random access to the elements is possible. Besides arrays, this assumptions is true in many other situations, e.g., for files of equal sized objects, or for tables in many database systems. In particular, this applies to column-oriented systems where random access is a basic mechanism needed to reconstruct result rows after applying filters to a small number of columns. Many other database systems provide random access. For example, the NoSQL system MongoDB by default creates an index on the unique object-ids.

To avoid special case discussions, we will henceforth assume that $n \leq N/2$. For the unusual case $n > N/2$, one can simply generate the $N - n < N/2$ elements that are *not* in the sample. When considering parallel algorithms, we use p to denote the number of processing elements (PEs), which we assume to be connected by a network. PEs are numbered $1..p$.

Algorithm S. Fan et al. [8] and Knuth [14] scan the range $1..N$ and generate a uniformly distributed random deviate for each element to decide whether it is sampled. For $N \gg n$, this is prohibitively slow and we are surprised that the algorithm still seems to be widely used, for example, by the GNU Scientific Library, GSL (function `gsl_ran_choose`, https://www.gnu.org/software/gsl/, version 2.2.1).

Algorithm H is a simple and efficient folklore algorithm that is very good for small n (see also Ahrens and Dieter [1]). The sample is kept in a hash table T that is initially empty. To produce the next sample element, it generates uniform deviates X from $1..N$. If $X \in T$, then it rejects X; otherwise, X is inserted into T. This algorithm runs in expected time $O(n)$. Note that T contains random numbers, and hence we can use a very simple hash function, such as extracting the most significant $\log n + O(1)$ bits from the key.[2] For $n \ll N$ the number of random deviates required is close to n, and we only need uniform deviates. This makes Algorithm H very fast for small n. For large n, however, most hash table accesses cause cache faults, slowing it down considerably.

Algorithm H could be parallelized. However, the hash table accesses then become global interactions between the PEs. The resulting overheads are even larger than the cache faults in the sequential algorithm and cause a severe bottleneck in distributed settings. One also has to be very careful if the resulting algorithm is supposed to be *deterministic*, i.e., the generated sample should be the same in repeated runs with the same seeds for the random number generators: race conditions in remote memory accesses or message delivery can easily lead to differences in the generated sample.

[1] We use the notation $a..b$ as shorthand for $\{a, \ldots, b\}$.
[2] In this article $\log x$ stands for $\log_2 x$.

Table 1. Overview of Abbreviations for the Algorithms

Abbrv.	Source	Time $O(\cdot)$	Space $O(\cdot)$	Mnemonic aid
S	[8]	N	1	scan
D	[27]	n	1	distance
B	[1]	n	n	Bernoulli
H	[15]	n	n	hash
R	Section 3.1	n	$\log n$	recursive
P	Section 3.2	$n/p + \log p$	$\log n$	parallel
SR	Section 4.1	n	1	sorted recursive

Algorithm D. Vitter [27] proposed an elegant sequential algorithm that generates the samples in sorted order without any need for auxiliary data structures. For generating the next sample, Algorithm D essentially generates an appropriate random deviate that specifies the number of positions to skip. Note that the random deviate changes in each step; using sophisticated techniques based on the rejection method, generating these random deviates can be done in constant expected time.

Algorithm B. Ahrens and Dieter [1] use the observation that taking a Bernoulli sample where each element of $1..N$ is sampled with probability $\rho \approx n/N$ yields a sample with $n' \approx n$ elements. If this sample is too big, then it can be repaired by removing $n' - n$ of the elements randomly. By choosing ρ somewhat larger than n/N, one can make the case $n' < n$ highly unlikely and simply restart the sampling process if it does occur. Bernoulli sampling can be implemented efficiently by generating geometrically distributed random deviates to determine how many elements to skip in each step. Algorithm B is faster than Algorithm D because generating geometric random deviates needs less arithmetic operations per element. In Section 4.7, we point out that it may be even more important that the parameters of the generated distribution remain the same, as this makes vectorization possible. A notable difference of Algorithm B to the aforementioned approaches is that elements are not generated online, i.e., we have an initial delay of $\Theta(n)$ before the first sample is generated.

Meng [18] gives a simple parallel sampling algorithm that works well when the data are stored on disk. However, it only works with high probability, needs to compute a random deviate for every input element, and needs communication between the PEs to perform a parallel sorting step of elements that may or may not be in the sample.

Table 1 summarizes the algorithm abbreviations used in this article.

3 DIVIDE-AND-CONQUER SAMPLING

Our central observation is that for any splitting position ℓ, the number of samples L from the left range $1..\ell$ is distributed hypergeometrically with parameters n (number of experiments), ℓ (number of success states), and N (universe size). Consequently, the number of samples from the right part $\ell + 1..N$ is $n - L$. We now apply this idea first to a sequential sampling algorithm in Section 3.1 and then give a parallelization in Section 3.2.

3.1 Sequential Divide-and-Conquer Sampling (Algorithm R)

Algorithm R in Figure 1 gives pseudocode for a sequential recursive divide-and-conquer algorithm based on the splitting approach described above. The range $1..N$ is split at $\ell = \lfloor N/2 \rfloor$. Then A is assigned to a recursively constructed sample of size L from $1..\ell$. B becomes a sample of size $n - L$

```
Function sampleR(n, N)
    if n < n_0 then return sampleBase(n, N)          // e.g. using algorithms H or D
    L:= hyperGeometricDeviate(n, ⌊N/2⌋, N)
    A:= sampleR(L, ⌊N/2⌋)
    B:= sampleR(n − L, N − ⌊N/2⌋)
    return A ∪ {x + ⌊N/2⌋ : x ∈ B}
```

Fig. 1. Algorithm R for (sequential) divide-and-conquer sampling without replacement.

from $\ell + 1..N$. After adding ℓ to the samples of B, we overall obtain the desired sample of size n from $1..N$.

The tuning parameter n_0 decides when to switch to the base case. When using Algorithm H, n_0 should be small enough so that the hash table fits into cache. Note that the resulting recursion tree has a size of at most $2n/n_0$. Hence the overall expected running time is $O(n)$, provided that we use a constant time algorithm for generating hypergeometric random deviates (e.g., Stadlober [24]) and a linear expected time algorithm for the base case.

3.2 Parallel Divide-and-Conquer Sampling (Algorithm P)

We now describe a parallel sampling algorithm that needs almost no communication. More precisely, each PE needs to know N and n but, otherwise, no communication is necessary. If we can assume that these values are known from the context, then no communication is needed at all. When these parameters are initially only known at PE 0, they can be broadcast with latency $O(\log p)$. In subsequent refinements, slightly larger amounts of communication are needed, but we will see that total communication overhead remains limited to $O(\log p)$.

For parallel sampling, we partition the range $1..N$ into p pieces. Let N_i denote the last element in the range associated with PE i, i.e., PE i generates the sample elements that lie in the subrange $N_{i-1}+1\,..\,N_i$ with $N_0 := 0$. The underlying idea of the parallelization is to adapt Algorithm R in a way such that $\lceil \log p \rceil$ levels of recursion split the original range $1..N$ into the subranges of each PE. Initially, all PEs work on the full range $1..N$. We use the PE numbers to gradually break this symmetry. After splitting the subproblem, each PE will follow only a single recursive call—the one whose range contains its local subrange.

Locally, when each PE works on its original range, we can use any sequential algorithm, but we have to be careful: On the one hand, PEs following the same path in this recursion tree have to generate the *same* random deviates to get a consistent result. On the other hand, random deviates generated in two *different* subtrees have to be independent. With true randomness (e.g., generated using a hardware random number generator [12]), this would require communication to distribute the right random values to the PEs (see also Section 4.4). However, using pseudorandomness (as most applications do) allows us to achieve the desired effect without any communication. The idea is to use a (high-quality) hash function h as source of pseudorandomness for generating hypergeometric deviates. In the subproblem for PEs $j..k$, the tth random deviate is the hash of the triple (j, k, t), i.e., $h((j, k, t))$. Figure 2 gives pseudocode where the function *hyperGeometricDeviate* is passed both h and $j..k$ in order to be able to use this technique. Within function *sampleLocally*, we can still use an ordinary generator of pseudorandomness that may have a better tradeoff between speed and quality than hashing. To break the symmetry between the PEs, we can seed it with $h(i)$ on PE i.

Another issue is that the PEs need access to the global element indices N_j. If the universe is evenly distributed between PEs (except for the last one if p does not divide n), then this is easy,

```
Function sampleP(n', j..k, i, h)
    if k − j = 1 then
        use h(i) to seed the local pseudorandom number generator
        M:= sampleLocally(n', N_i − N_{i−1} + 1)          // e.g. using algorithms H, D, or R
        return {N_{i−1} + x : x ∈ M}
    m:= ⌊(j+k)/2⌋                                          // middle PE number
    L:= hyperGeometricDeviate(n', N_m − N_j + 1, N_k − N_j + 1, j..k, h)
    if i ≤ m then return sampleP(L, j..m, i, h)
    else return sampleP(n' − L, m + 1..k, i, h)
```

Fig. 2. Algorithm P for sampling n' elements on PEs $j..k$, where $i \in j..k$ is the PE executing the function. The initial call on PE i is $sampleP(n, 1..p, i, h)$.

as we simply have $N_j = j\lceil n/p \rceil$ for $j < p$ and $N_p = N$. Refer to Section 4.3 for the case of uneven distribution of the universe.

We obtain the following running time for Algorithm P.

Theorem 3.1. *If* $\max_i (N_i - N_{i-1}) = O(N/p)$, *then Algorithm P runs in time* $O(n/p + \log p)$ *with high probability.*[3]

Proof. First note that the bound holds when we only calculate with expectations. Each PE generates $\leq \lceil \log p \rceil$ hypergeometric random deviates and $O(n/p)$ samples in expectation.

However, we also have to take into account rare cases that could slow down computation on some PE wthat would then lead to a large overall execution time. Three issues have to be considered: deviations in the number of samples per PE, deviations in the time needed to generate the random deviates, and running time fluctuations within function *sampleLocally*.

The number of samples generated by one PE has a hypergeometric distribution. We exploit that this distribution spreads the elements more evenly than a binomial distribution [19, Theorem 3.3] and analyze the simpler situation when each sample is independently assigned to PE i with probability $(N_i - N_{i-1} + 1)/N = O(1/p)$. We thus have a classical balls-into-bin situation that can be analyzed using Chernoff bounds. The particular calculations needed here have been done in Reference [19, Theorem 3.7] and yield exactly what we need – $O(\log p)$ samples with high probability when $n = O(p \log p)$ and $O(n/p)$ samples with high probability when $n = \Omega(p \log p)$.

Fast algorithms for generating hypergeometric deviates [24] are often based on a rejection method, i.e., they generate a constant number of uniform deviates, perform a constant amount of computation, and then perform a test that succeeds with constant probability. If the test fails, then an independent new trial is performed. Hence, the running time of the generator can be bounded by a constant times a geometrically distributed random variable. Since each PE generates $\lceil \log p \rceil$ hypergeometric deviates, we need a tail bound on the sum of a logarithmic number of exponential deviates with constant expectation. By Lemma 3.2 below this sum is $O(\log p)$ with high probability. By choosing the parameter c in the definition of "with high probability" by one larger than what we want to show in the overall theorem, we can accommodate the fact that the running time on the slowest PE determines the overall running time—the probability that any of the p PEs is slower than claimed is bounded by $p \cdot p^{-(1+c)} = p^{-c}$.

When using Algorithm D for generating local samples, we can use a similar argument as above—the running time for generating each sample is bounded by a geometrically distributed random variable so that large deviations from the expectation are unlikely. When using Algorithm H,

[3]We say a statement is true with high probability if it is true with probability at least $1 - p^{-c}$ for any constant c.

the details of the analysis depend on the tails of the running time distribution of the hash table, but we will get the required short tails of the running time distribution if we allocate enough space ($O(n/p + \log p)$) for this table. When using Algorithm R, additional hypergeometric random deviates are generated, but the argument with the geometrically distributed running time again holds. □

LEMMA 3.2. *Consider $k = \Omega(\log p)$ independent geometrically distributed random variables X_i with constant expectation. Then their sum Y is in $O(k)$ with high probability.*

PROOF. By considering the geometric random variable with largest expectation, we can assume wlog that the variables are identically distributed. The sum of k independent, identically distributed random variables has a negative binomial distribution. This situation can be analyzed using Chernoff bounds by exploiting the relation between binomial distribution and negative binomial distribution. We obtain the following:

$$\Pr[Y > a\mathrm{E}[Y]] \leq e^{-\frac{ak(1-1/a)^2}{2}} \leq p^{-\Omega(1)\frac{a(1-1/a)^2}{2}} \leq p^{-c}.$$

The first "$\leq$" follows a derivation of Brown [6]. The second "$\leq$" exploits that $k = \Omega(\log p)$, and the third inequality chooses a sufficiently large constant a for any particular choice of the constant c from the definition of "with high probability." □

4 GENERALIZATIONS

4.1 Generating Output in Sorted Order and Online

Note that Algorithm R can easily output the elements in sorted order provided that the base case algorithm generates the samples in sorted order. This is certainly the case when using Algorithm D, and we can also adapt Algorithm H for this purpose. For example, we can maintain the invariant that the samples in the hash table are sorted. This is possible since we use the most significant bits to address the table. We only have to ensure that colliding elements are also sorted. Rather than appending an inserted element k to the end of a cluster of colliding elements, we skip elements smaller than k and then shift the cluster elements larger than k one position to the right. This makes handling clusters of colliding elements somewhat slower, but the overall overhead is small since the clusters are small (expected constant size).

Alternatively, we can insert into the hash table normally—ignoring the ordering of the keys—and sort the hash table afterwards. Since the sorting order is the same as the hash function value, the only thing we have to do is to scan the hash table and sort clusters of colliding elements. This leads to a linear time algorithm since the clusters are small.

It is also easy to modify Algorithm R to generate samples online with constant expected delay between generated samples. We can modify the divide-and-conquer step to split off a range of size $\lceil N \cdot n_0/n \rceil$. Using this splitting in an iterative fashion, we generate samples in batches of expected size of approximately n_0. This takes time $O(n_0)$ per batch, i.e., constant time if n_0 is a constant.

The same techniques can be used in a parallel setting. Then each PE generates the elements of its designated subrange of $1..N$ in sorted order.

4.2 Load Balancing

Algorithm R implicitly assumes that all PEs are equally fast. However, for various reasons, this may not be the case. For example, we might work with heterogeneous cloud resources, there might be other jobs (or operating system services) slowing down some PEs, or uneven cooling might imply different clock frequencies for different PEs. In these cases, the slowest PE would slow down the overall computation. This problem can be solved with standard load balancing techniques. We

split $1..N$ into $p' \gg p$ jobs (subranges) and use a load balancing algorithm to dynamically assign jobs to PEs.

The most widely used load balancing method for such problems uses a centralized master PE to assign jobs to PEs. Unfortunately, this increases the running time from $O(n/p + \log p)$ (Theorem 3.1) to $O(n/p + p)$. A more scalable approach is *work stealing* [5, 9]. To employ this approach, we instantiate the concept of a *tree shaped computation* [20]: We conceptually split the work into very fine-grained *atomic* jobs corresponding to ranges of sample values that are expected to contain a constant number of samples (say, n_0). However, initially these jobs are coalesced into p (meta) jobs of about equal size. Now each PE sequentially works on its meta job, one atomic job at a time. Idle PEs ask random other PEs to split their range of unfinished atomic jobs in half, delivering one half to the idle PE. Note that both splitting off the next atomic job and splitting the remaining range of atomic jobs in half can be done in constant expected time using the division strategy from Algorithm P. The generic analysis in Ref. [20] then yields the same asymptotic running time as in Theorem 3.1.

At least on shared memory machines, this asymptotically scalable load balancing is easy to implement since it is part of widely used tools such as the C++ standard library [23], the Intel Thread Building Blocks, or Cilk [4].

4.3 Uneven Distribution of the Sampled Universe

When we sample from a set of elements distributed over PEs connected by a network, we may not want to load balance. Rather, we want to use the *owner computes* paradigm—each PE computes those samples that stem from its local subset of elements.[4] In this situation, each PE i initially only knows its local number of elements L_i.

We address this situation by arranging the PEs into a binomial tree [26]. Let the PEs be numbered $0..p-1$ now to facilitate binary arithmetic. If the binary representation of the PE number i ($\lceil \log p \rceil$ bits) contains k trailing zeroes, then it is connected with PEs $i + 2^j$ for $j \in 0..k-1$ if $i + 2^j < p$. The connections for each value of j form one level of a binary tree, as shown in the example in Figure 3. At level $j \in 0..\lceil \log p \rceil$, we get (maximal) subtrees spanning PEs $2^j a.. \min(2^j a + 2^j - 1, p-1)$ for $a = 0..p/2^j - 1$. In an upward pass, iterating from $j = 0$ upwards, we compute the sum of the L-values in each of these subtrees. For an inner node, let L_ℓ and L_r denote the partial sums for its left and right subtrees, respectively.

Now the number of samples in each subtree is computed in a top-down fashion. The root knows that it has to generate $n' = n$ samples. Other nodes receive their n' value from their parent. An inner node uses a hypergeometric distribution with parameters n, L_ℓ, and $L_\ell + L_r$ to split its n' samples into $n' = n_\ell + n_r$. Then n_ℓ is used for the next smaller subtree locally, while n_r is passed to the right child as the number of samples to be generated there. To generate independent random values everywhere, the subtree representing PEs $a..b$ can use this range as an input for the hash function h from Algorithm P.

4.4 Using True Randomness

Now let us assume that each PE has access to some independent physical source of truly random values. In this case, we can use the algorithm from Section 4.3, since it makes every random decision only once and explicitly passes the resulting information to other PEs.

[4] In a hybrid setting, where several shared memory machines are connected by a network, we could still apply load balancing on each shared memory machine.

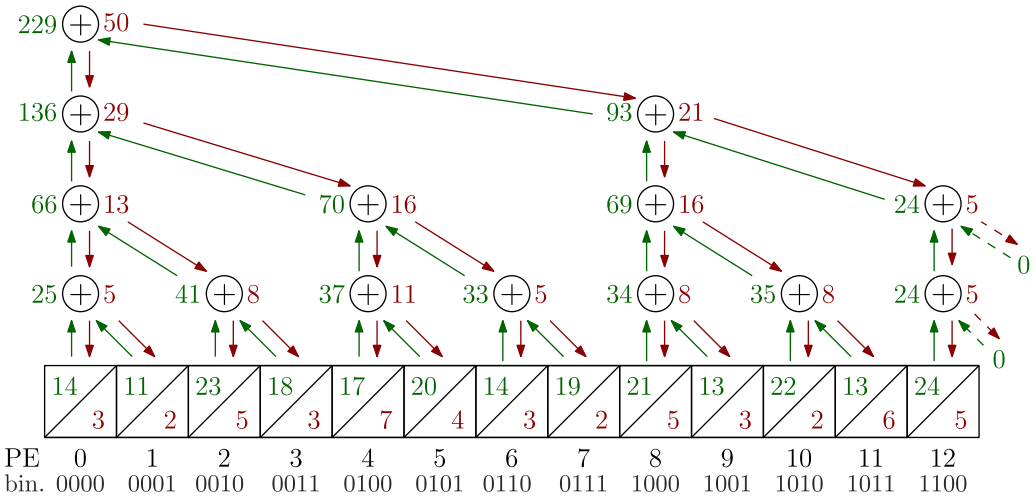

Fig. 3. Assigning $n = 50$ samples to 13 PEs, with $N = 229$ elements. Element counts (L-values, in green, on the nodes' left side) are added bottom-up, sample counts (n', in red, right side) are assigned top-down.

4.5 Deterministic Results

For fixed p and h, Algorithm P deterministically and reproducibly generates the same sample every time, which is important to make software using the algorithm predictable, reliable, and testable. If we even want the result to be independent of p, then we can use the load balancing method from Section 4.2. In this case, we generate $p' \gg p$ jobs regardless of the actual number of PEs used and then assign the jobs to the PEs (possibly even statically, $\lceil p'/p \rceil$ consecutive jobs for each PE).

4.6 Sampling with Replacement in Various Spaces

Algorithms R and P are easy to adapt to sampling with replacement. The only thing that changes is that the hypergeometric distribution for the divide-and-conquer step has to be replaced by a binomial distribution. Note that this is not restricted to sampling from the one-dimensional discrete range $1..N$. We can also uniformly sample from continuous or higher-dimensional sets as long as we can bipartition the space. For example, for generating random points in a rectangle, we can subsequently bisect this rectangle into smaller and smaller rectangles up to some base case. To match the size of these base objects to the number of PEs, it might be useful to generate $K \gg p$ base objects and to use some kind of load balancing to map base objects to PEs. This works similar to the load balancing methods from Section 4.2.

4.7 Relation to Bernoulli Sampling

We want to point out that Bernoulli sampling and sampling without replacement are almost equivalent in the sense that they can emulate each other efficiently. On the one hand, Bernoulli sampling with success probability ρ can be implemented by sampling without replacement if we can first determine how many elements n are sampled by Bernoulli sampling. This number follows a binomial distribution with parameters N and ρ. Then we can use sampling without replacement to choose the actual elements. On machines with slow floating point arithmetics, e.g., microcontrollers, this approach might be faster than generating skip values from a geometric distribution, which requires evaluating logarithms.

On the other hand, Algorithm B [1] generates n samples without replacement by "repairing" a Bernoulli sample. For this article, it is important that Bernoulli sampling can also be parallelized in several ways. We can independently apply Bernoulli sampling to subranges of $1..N$. This is the method of choice for distributed memory machines since it requires no communication. On a shared memory machine, we can also generate an array of $(1 + o(1))\rho N$ independent, geometrically distributed random deviates and compute their prefix sums. The values up to N denote the sample. A practically important observation is that the operation needed for this approach has no conditional branches or random memory accesses and hence can be implemented on single instruction multiple data (SIMD) units of modern CPUs or on GPUs.

To parallelize Algorithm B, we can use it as base case of Algorithm P. We can also use parallel Bernoulli sampling and then use Algorithm P in the repair step.

5 IMPLEMENTATION DETAILS

We have implemented algorithms D, H, R, P, and B using C++.[5] Refer to Table 1 for a summary of the abbreviations. We use Spooky Hash[6] as a hash function that generates seeds for initializing the Mersenne twister [17] pseudorandom number generator for uniform deviates.

Algorithm D has been translated literally from the description in Ref. [27].

Algorithm H uses hashing with linear probing [15] using a power of two as table size. We use two variants for obtaining the entries of the table and for emptying it. The default is to record the positions of inserted elements on a stack. This way, we can retrieve and reset the table elements without having to consider empty entries. In turn, this allows us to make the table size m significantly larger than the final number of entries n to speed up table accesses. This does not work when we want to output table entries in sorted order. Here we omit the stack and explicitly scan the table at the end. Furthermore, we allocate n additional table entries to the right so that it becomes unnecessary to wrap around when an insertion probes beyond the mth table entry. Otherwise, wrapping around could destroy the globally sorted order between clusters (see Section 4.1).

Algorithm R uses Algorithm H as the base case sampler (*sampleBase* in the pseudocode of Figure 1). We do this because Algorithm H is faster than Algorithm D for small subproblems where the hash table fits into cache. This will always be the case if n_0 is chosen appropriately (we use $n_0 = 2^9$ and $m = 2^{12}$). To generate hypergeometric random deviates, we use the `stocc` library,[7] which uses a Mersenne twister internally.

Algorithm P on Blue Gene/Q is parallelized using MPICH 1.5 on gcc 4.9.3. It uses Algorithm R with parameters $n_0 = 2^8$ and $m = 2^{11}$ as local sampling algorithm.

Algorithm B uses Algorithm R for selecting samples to be removed in the repair step. Geometric random deviates are generated using the C++ standard library (`std::geometric_distribution` and `std::mt19937_64`).

We implemented two further variants of Algorithm B. One targets SIMD parallelism within a single CPU-core. The other uses NVIDIA GPUs. The CPU-SIMD version performs best when restricting arithmetics to 32 bits. Therefore we use a smaller maximal universe size of $N = 2^{30}$ there. This version uses the Intel Math Kernel Library MKL v11.3 [13] to generate geometric deviates. Prefix sums are computed by a manually tuned routine using SSE2 instructions through compiler intrinsics.

[5] https://github.com/lorenzhs/sampling/tree/toms.
[6] http://www.burtleburtle.net/bob/hash/spooky.html, version 2.
[7] http://www.agner.org/random/, version 2014-Jun-14.

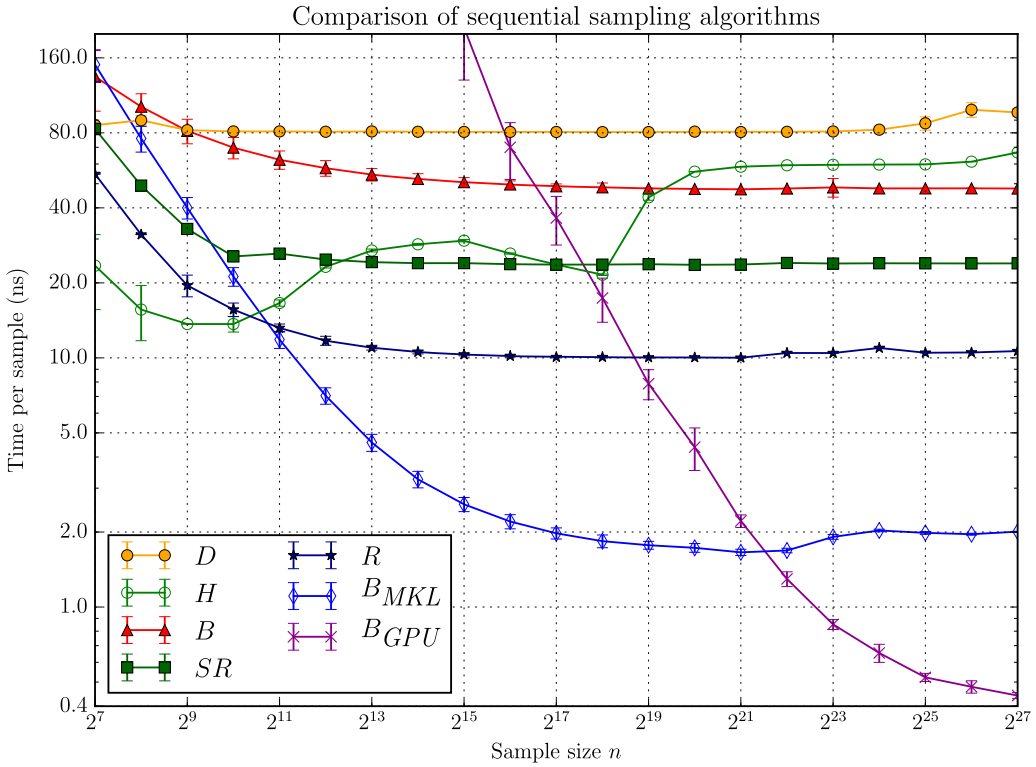

Fig. 4. Running time per sample for the sequential algorithms H, D, R, and B. The bars show the standard deviation. The number of repetitions for each algorithm is $2^{30}/n$. For Algorithm R, we use $n_0 = 2^{10}$. *SR* is Algorithm R with sorted output. B_{MKL} and B_{GPU} are non-portable vectorized implementations of Algorithm B for CPUs using Intel's Math Kernel Library (MKL) and NVIDIA GPUs using CUDA, respectively.

The GPU version uses CUDA 7.5, the cuRAND library[8] for generating geometric random deviates, and the Thrust library[9] for computing prefix sums. Thus, most of the work can actually be delegated to libraries tuned by the vendor. Unfortunately, the repair step, albeit requiring only sublinear work, is difficult to do on the GPU. Therefore, it is partially delegated to the CPU. There are various ways to accomplish this, but the key point is to do it in a way such that the sample does not need to be transferred to the CPU. Our solution first uses a parallel GPU pass over the sample to count the number n' of prefix sum values $< N$ (see Section 2). Only the single value n' needs to be transferred to the CPU. The CPU then uses Algorithm R to generate $n' - n$ samples from the range $0..n' - 1$. These samples are transferred to the GPU, which marks the appropriate positions in the sample array for removal. Finally, the sample array is compacted using the Thrust function `copy_if`.

6 EXPERIMENTS

Figure 4 compares the performance of the sequential Algorithms D, H, R, and B. Refer to Table 1 for a summary of the abbreviations. These experiments were conducted on a single core of

[8]http://docs.nvidia.com/cuda/curand/, v7.5.
[9]https://developer.nvidia.com/thrust, v1.7.0.

a dual-socket Intel Xeon E5-2670 v3 system with 128GiB of DDR4-2133 memory, running Ubuntu 14.04. The code was compiled with GNU g++ in version 6.2 using optimization level `fast` and `march=native`. We report results for universe size $N = 2^{50}$ and varying n. The number of repetitions was $2^{30}/n$ to achieve equal work for every n. We see that Algorithm H is very fast for small n, but its performance degrades as n grows and the hash table exceeds the cache size. Our new Algorithm R is similarly fast for small n, but the time per sample remains constant as n grows. Thus, it is up to 5 times faster than Algorithm H for very large n. The performance of Algorithm D is also independent of n but worse than Algorithm R by a factor of 7. A variant of Algorithm R (SR) that generates samples online and in sorted order is still 3.4 times faster than Algorithm D. The portable implementation of Algorithm B (labeled B in Figure 4) is faster than Algorithm D but cannot compete with Algorithm R.

This picture changes when looking at tuned architecture specific implementations of Algorithm B. The CPU version (label B_{MKL}) is up to 6 times faster than Algorithm R for large n. For very large n, the GPU version, B_{GPU}, running on an NVIDIA GeForce GTX 980 Ti graphic card, is 4.5 times faster. However, it should be noted that a single core of a Xeon E5-2670 v3 uses much less power than an entire GTX 980 Ti—the entire Xeon PE with 12 cores uses about half the power of the graphics card.

Our experiments clearly confirm our expectation that offloading sampling to the GPU for further processing on the CPU is not worthwhile, as the time for transferring the samples from GPU to CPU memory (not pictured in Figure 4) dwarfs the time to take the sample—including transfer, a single core of a modern CPU can generate the samples equally fast. However, it also shows that fast sampling is possible for GPU applications, i.e., if the samples are required on the GPU for further processing.

It is also worth looking at the individual components of the running time of the GPU implementation (we consider the case of $n = 2^{27}$ as an example). The CPU portion and data transfer account for 13.3% of the total running time, while 86.7% are spent on computation on the GPU. This time, in turn, is split up as follows. Generating geometrically distributed random numbers using cuRAND takes 25.0% of the computation time, and calculating a prefix sum over the elements using the Thrust function `inclusive_scan` takes another 36.7%. Counting the number of elements $< N$ with `count_if` takes 6.9%. While the time for marking the elements selected by the CPU is negligible at 0.2%, the following compaction with the Thrust function `copy_if` takes another 31.0%.

Algorithm P. Figure 5 shows a so-called weak scaling experiment on JUQUEEN, a distributed memory machine. It shows the running time of Algorithm P when keeping local input size n/p constant, measured for different values of this ratio. JUQUEEN is an IBM Blue Gene/Q machine, demonstrating the portability of our code. We used the maximum number of 16 cores per node for these experiments. Performance per core is an order of magnitude lower than on the Intel CPU used for our sequential experiments. A factor of four is more typical for other applications considering the lower clock frequency, older technology, and lower number of transistors used. The remaining factor of 2–3 is mostly due to the fact that our random number generator, a SIMD-oriented Mersenne Twister, contains optimizations to make use of the SSE2 units of Intel CPUs. However, it does not have similar optimizations for the QPX instructions of Blue Gene/Q, thus reverting to scalar code. This is compounded further by the lack of autovectorization for Blue Gene/Q in gcc.

On the positive side, we see that the code scales almost perfectly for sufficiently large values of n/p. For the smallest tested value of n/p, 4096, we see a linear increase in running time with an exponential increase in p. This is consistent with the asymptotic running time of $n/p + \log p$.

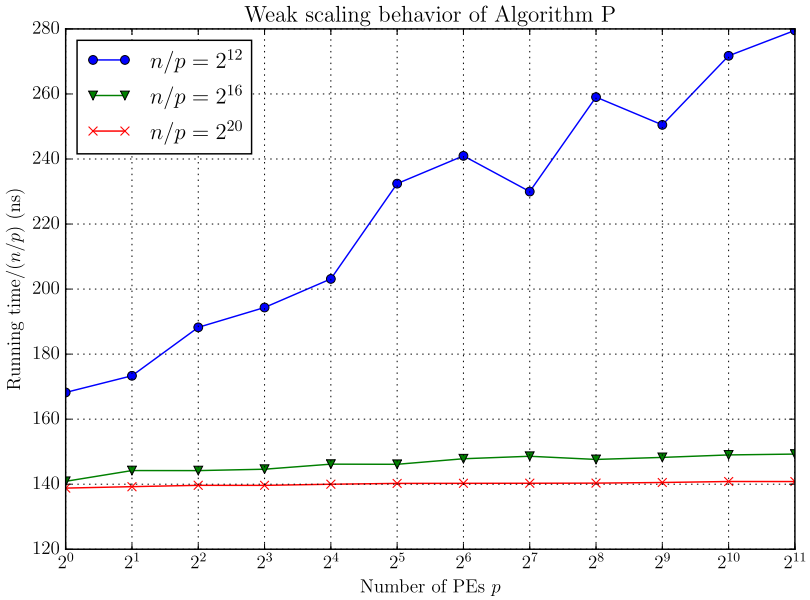

Fig. 5. Running time for generating n samples on p PEs for different values of n/p using Algorithm P with Algorithm R as local sampler, using $n_0 = 2^8$. The number of repetitions for each value of n/p is $2^{30} \cdot p/n$.

7 CONCLUSIONS

We find it surprising that the seemingly trivial problem of random sampling requires such a diverse set of algorithmic techniques. Moreover, the features of modern computer architectures entail that no single approach is universally best. When n is very small, Algorithm H is both simple and efficient, but for larger n it becomes cache inefficient. This problem can be overcome by using it for the base case of Algorithm R. A slight generalization of Algorithm R allows for parallelization (Algorithm P). Since this requires no or almost no communication, it is suitable for many parallel models of computation, such as shared memory, distributed memory, or cloud computing. Only some details like load balancing and adaptation to nonuniform data distribution require communication.

On the other hand, we see no reason to continue using Algorithm S. It is fast (only) if N/n is a small constant, but we doubt that it can ever outperform Algorithm R, which needs at most half the number of uniform deviates. In particular, for small N/n we could use a variant of Algorithm H for the base case that uses the key directly to index the table of sampled elements. This avoids the need for handling collisions between samples.

The main point in favor of Algorithm D is that it generates the samples in sorted order and works in an online fashion, i.e., the expected time between generating samples is constant. With the iterative version of Algorithm R described in Section 4.1, we can achieve the same effect but with a more flexible tradeoff between maximum latency between samples and the average cost per sample. From that perspective, our divide-and-conquer technique is a generalization of Algorithm D that allows faster processing and parallelization. Actual real time guarantees of deterministic constant time between subsequent samples seem to be an open problem and neither Algorithm R nor D can offer such guarantees.

Algorithm B is useful because it allows for vectorization. Hence, on architectures with fast arithmetics, a tuned version of Algorithm B can outperform Algorithm R. However, this comes at the

cost of reduced portability and that samples cannot be generated in an online fashion—it is only after the repair step that we know which samples survive.

To illustrate the usefulness of fast sampling algorithms, we mention a few applications. Our algorithms are most useful in settings where fast random access (by index) to the elements is possible, e.g., when dealing with fixed-size records or in main-memory-based databases such as SAP HANA [22], SAP HANA Vora [11], or EXASOL (http://www.exasol.com/). Generating a random graph in the $G(n, m)$ and $G(n, p)$ model of Erdős and Rényi [7] is equivalent to sampling from the $n(n-1)/2$ possible edges. Sampling is performed without replacement for $G(n, m)$ and Bernoulli sampling is used for $G(n, p)$. In our library for generating massive graphs [10, 16], we successfully use the sampling algorithms described here. In the same work, we also use sampling with replacement to generate point sets for several families of random geometric graphs (proximity-based 2D, 3D; Delaunay 2D, 3D; and hyperbolic). Experiments indicate good performance for up to 2^{18} PEs. Sample sort [2, 3] is a successful example of a parallel sorting algorithm that splits its input based on a random sample. With Algorithm P, this is now possible with very low overhead and without resorting to simplified sampling models, which often complicate the analysis and reduce sampling quality. This is also an example where scalability matters, i.e., where a bound of $O(n/p + \log p)$ is much better than $O(n/p + p)$.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the Gauss Centre for Supercomputing (GCS) for providing computing time through the John von Neumann Institute for Computing (NIC) on the GCS share of the supercomputer JUQUEEN [25] at Jülich Supercomputing Centre (JSC). GCS is the alliance of the three national supercomputing centres HLRS (Universität Stuttgart), JSC (Forschungszentrum Jülich), and LRZ (Bayerische Akademie der Wissenschaften), funded by the German Federal Ministry of Education and Research (BMBF) and the German State Ministries for Research of Baden-Württemberg (MWK), Bayern (StMWFK), and Nordrhein-Westfalen (MIWF).

REFERENCES

[1] J. H. Ahrens and U. Dieter. 1985. Sequential random sampling. *ACM Trans. Math. Softw.* 11, 2 (Jun. 1985), 157–169.

[2] M. Axtmann, T. Bingmann, P. Sanders, and C. Schulz. 2015. Practical massively parallel sorting. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'15).*

[3] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. 1991. A comparison of sorting algorithms for the connection machine CM-2. In *Proceedings of the 3rd ACM Symposium on Parallel Algorithms and Architectures (SPAA'91).* 3–16.

[4] R. D. Blumofe, C. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. 1995. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'95).* ACM, 207–216.

[5] R. D. Blumofe and C. E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46, 5 (1999), 720–748.

[6] D. G. Brown. 2011. How I wasted too long finding a concentration inequality for sums of geometric variables. Retrieved from https://cs.uwaterloo.ca/~browndg/negbin.pdf.

[7] P. Erdös and A. Rényi. 1959. On random graphs, I. *Publ. Math. (Debrecen)* 6 (1959), 290–297.

[8] C. T. Fan, M. E. Muller, and I. Rezucha. 1962. Development of sampling plans by using sequential (item by item) selection techniques and digital computers. *J. Am. Statist. Assoc.* 57, 298 (1962), 387–402.

[9] R. Finkel and U. Manber. 1987. DIB—A distributed implementation of backtracking. *ACM Trans. Prog. Lang. Syst.* 9, 2 (Apr. 1987), 235–256.

[10] D. Funke, S. Lamm, P. Sanders, C. Schulz, D. Strash, and M. von Looz. 2018. Communication-free massively distributed graph generation. In *Proceedings of the 32nd IEEE International Parallel & Distributed Processing Symposium (IPDPS'18).* To appear.

[11] A. Goel, J. Pound, N. Auch, P. Bumbulis, S. MacLean, F. Färber, F. Gropengiesser, C. Mathis, T. Bodner, and W. Lehner. 2015. Towards scalable real-time analytics: An architecture for scale-out of OLxP workloads. *Proc. VLDB Endow.* 8, 12 (2015), 1716–1727.

[12] Intel. 2012. *Intel Digital Random Number Generator (DRNG): Software Implementation Guide*. Intel. Retrieved from https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide.
[13] Intel. 2015. *Intel Math Kernel Library v11.3*. Intel. Retrieved from https://software.intel.com/en-us/mkl-reference-manual-for-c.
[14] D. E. Knuth. 1981. *The Art of Computer Programming—Seminumerical Algorithms*, 2nd ed. Vol. 2. Addison–Wesley.
[15] D. E. Knuth. 1998. *The Art of Computer Programming—Sorting and Searching*, 2nd ed. Vol. 3. Addison–Wesley.
[16] S. Lamm. 2017. *Communication Efficient Algorithms for Generating Massive Networks*. Master's thesis. Karlsruhe Institute of Technology.
[17] M. Matsumoto and T. Nishimura. 1998. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.* 8, 1 (1998), 3–30.
[18] X. Meng. 2013. Scalable simple random sampling and stratified sampling. In *Int. Conf. on Machine Learning (ICML'13)*. 531–539.
[19] P. Sanders. 1996. *Lastverteilungsalgorithmen für parallele Tiefensuche*. Dissertation. Universität Karlsruhe.
[20] P. Sanders. 2002. Randomized receiver initiated load balancing algorithms for tree shaped computations. *Comput. J.* 45, 5 (2002), 561–573.
[21] P. Sanders, S. Schlag, and I. Müller. 2013. Communication efficient algorithms for fundamental big data problems. In *Proceedings of the IEEE International Conference on Big Data*. 15–23.
[22] V. Sikka, F. Färber, A. Goel, and W. Lehner. 2013. SAP HANA: The evolution from a modern main-memory data platform to an enterprise application platform. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1184–1185.
[23] J. Singler, P. Sanders, and F. Putze. 2007. MCSTL: The multi-core standard template library. In *Proceedings of the 13th Int. Euro-Par Conf. (LNCS)*, Vol. 4641. Springer, 682–694.
[24] E. Stadlober. 1990. The ratio of uniforms approach for generating discrete random variates. *J. Comput. Appl. Math.* 31, 1 (1990), 181–189.
[25] M. Stephan and J. Docter. 2015. Jülich supercomputing centre. JUQUEEN: IBM Blue Gene/Q supercomputer system at the Jülich Supercomputing Centre. *Journal of Large-scale Research Facilities* A1 (1 2015), 1–5.
[26] H. Sullivan and T. R. Bashkow. 1977. A large scale, homogeneous, fully distributed parallel machine, I. In *Proceedings of the 4th Annual Symposium on Computer Architecture (ISCA'77)*. ACM, New York, NY, 105–117.
[27] J. S. Vitter. 1984. Faster methods for random sampling. *Commun. ACM* 27, 7 (July 1984), 703–718.