Parallel Weighted Random Sampling

Lorenz Hübschle-Schneider
Karlsruhe Institute of Technology, Germany
huebschle@kit.edu

Peter Sanders
Karlsruhe Institute of Technology, Germany
sanders@kit.edu

Abstract

Data structures for efficient sampling from a set of weighted items are an important building block of many applications. However, few parallel solutions are known. We close many of these gaps both for shared-memory and distributed-memory machines. We give efficient, fast, and practicable algorithms for sampling single items, k items with/without replacement, permutation, subset sampling, and reservoir sampling. Our output sensitive algorithm for sampling with replacement also improves the state of the art for sequential algorithms.

2012 ACM Subject Classification Theory of computation → Sketching and sampling; Theory of computation → Parallel algorithms; Theory of computation → Data structures design and analysis

Keywords and phrases parallel algorithm, sampling, PRAM, communication efficient algorithm

Supplement Material The code is available under the GPL at https://github.com/lorenzhs/wrs.

1 Introduction

Weighted random sampling asks for sampling items (elements) from a set such that the probability of sampling item i is proportional to a given weight w_i. Several variants of this fundamental computational task appear in a wide range of applications in statistics and computer science, *e.g.,* for computer simulations, data analysis, database systems, and online ad auctions (see, *e.g.,* Motwani et al. [23], Olken et al. [25]). Continually growing data volumes ("Big Data") imply that the input sets and even the sample itself can become large. Since actually processing the sample is often fast, sampling algorithms can easily become a performance bottleneck. Due to the hardware developments of the last years, this means that we need *parallel algorithms* for weighted sampling. This includes *shared-memory* algorithms that exploit current multi-core processors, and *distributed algorithms* that split the work across multiple machines without incurring too much overhead for communication.

However, there has been surprisingly little work on parallel weighted sampling. This paper closes many of these gaps. Table 1 summarizes our results on the following widely used variants of the weighted sampling problem. We process the input set $A = 1..n$ on p processing elements (PEs) where $i..j$ is a shorthand for $\{i, \dots, j\}$. Item i has weight w_i and $W := \sum_{i=1}^{n} w_i$. Define $u := \log U$ where $U := w_{\max}/w_{\min} := \max_i w_i / \min_i w_i$.

WRS–1: Weighted sampling of *one item* (equivalent to WRS–R and WRS–N for $k = 1$).

WRS–R: Sample k items from A *with replacement*, *i.e.,* the samples are independent and for each sample X, $\mathbf{P}[X = i] = w_i/W$. Let $s = |S| \leq k$ denote the number of *different* items in the sample S. Note that we may have $s \ll k$ for skewed input distributions.

WRS–N: Sample k pairwise unequal items $s_1 \neq \cdots \neq s_k$ *without replacement* such that $\mathbf{P}[s_l = i] = w_i/(W - w_{s_1} - \cdots - w_{s_{\ell-1}})$.

WRP: Permute the elements with the same process as for WRS–N using $k = n$.

WRS–S: Sample a *subset* $S \subseteq A$ where $\mathbf{P}[i \in S] = w_i \leq 1$.

WRS–B: *Batched reservoir sampling.* Repeatedly solve WRS–N when batches of new items arrive. Only the current sample and batch may be stored. Let b denote the batch size.

	Shared Memory					Distributed		
		Preprocessing		Query			Preprocessing	Query
Problem	§	Work	Span	Work	Span	§	Time	Time
WRS–1	4	n	$\log n$	—	1	5	$\frac{n}{p} + \alpha \log p$	α
WRS–R	6	$\text{isort}^*_u(n)$		$s + \log n$	$\log n$	6.1	$\text{isort}^1_u(\frac{n}{p}) + \alpha \log p$	$\frac{s}{p} + \log p$
WRS–N	7	$\text{isort}^*_u(n)$		$k + \log n$	$\log n$	7.1	$\text{isort}^1_u(\frac{n}{p}) + \alpha \log p$	$\frac{k}{p} + \alpha \log^2 n$
WRS–N						7.1	$\frac{n}{p} + \beta u + \alpha \log p$	$\frac{k}{p} + \alpha \log n$
WRP	8	—	—	$\text{isort}^*_{n(u+\log n)}(n)$		8	—	$\text{isort}^*_{n(u+\log n)}(n)$
WRS–S	9	n	$\log n$	$s + \log n$	$\log n$	9	$\frac{n}{p} + \log p$	$\frac{s}{p} + \log p$
WRS–B	—	—	—	—	—	10	—	$b + b^* \log(b^* + k) + \alpha \log^2 kp$

Table 1 Overview of our results. Distributed results assume random distribution of inputs. All results are expected and asymptotic. Input size n, output size s, sample size k, startup latency of point-to-point communication α, time for communicating one machine word β, log-weight ratio $u = \log U = \log w_{\max}/w_{\min}$, mini-batches of b items per PE, at most b^* of which below insertion threshold. The complexity of sorting n integers with keys from $0..x$ is $\text{isort}_x(n)$ (isort^* = parallel, isort^1 = sequential).

When applicable, our algorithms build a data structure once which is later used to support fast sampling queries. Most of the algorithms have linear work and logarithmic (or even constant) latency. Neither competitive parallel algorithms nor more efficient sequential algorithms are known. The distributed algorithms are refinements of the shared-memory algorithms with the goal to reduce communication costs compared to a direct distributed implementation of the shared-memory algorithms. As a consequence, each PE mostly works on its local data (the owner-computes approach). Communication – if at all – is only performed to coordinate the PEs and is sublinear in the local work except for extreme corner cases. The owner-computes approach introduces the complication that differences in local work introduce additional parameters into the analysis that characterize the local work in different situations (*e.g.,* the last line of Table 1). The summary in Table 1 therefore covers the case when items are randomly assigned to PEs. This simplifies the exposition and is actually an approach that one can sometimes take in practice.

First, in Section 2, we review the models of computation used in this paper as well as known techniques we are building on. We discuss additional related work in Section 3.

In Section 4, we develop a parallelization of *alias tables*, the most widely used data structure for Problem WRS–1. It seems to be a difficult open problem to actually compute the same data structure in parallel. Yet slightly generalizing it allows for a fast parallel algorithm that supports queries with a single memory probe. In Section 4.1 we describe a variant that needs only $\mathcal{O}(n)$ *bits* of additional space. Section 5 describes distributed variants where each PE builds a local alias table. In addition, we build an alias table for p *meta-items* that represent the total weight of each PE. Queries become a two-stage process where the meta-items' alias table is queried to determine the PE responsible for computing the actual result.

Sampling k items with replacement (Problem WRS–R) seems to be trivially parallelizable with an alias table. However this does not lead to a communication-efficient distributed algorithm and we can generally do better for skewed input distributions where the number of *distinct* output elements s can be much smaller than k. Section 6 develops such an algorithm for PRAMs which is however also interesting as a sequential algorithm. The algorithm combines three previous techniques and requires a nontrivial analysis. Section 6.1 adapts the algorithm to a distributed setting where a communication-efficient construction algorithm

and even a communication-free query is achieved.

Section 7 employs the algorithm for Problem WRS–R to solve Problem WRS–N. The main difficulty here is to estimate the right number of samples with replacement to obtain a sufficient number of distinct samples. Then an algorithm for WRS–N without preprocessing is used to reduce the "weighted oversample" to the desired exact output size.

It is well known that the weighted permutation Problem WRP can be reduced to sorting (see Section 2.3). We show in Section 8 that this is actually possible with linear work by appropriately defining the (random) sorting keys so that we can use integer sorting with a small number of different keys. Since previous linear-time algorithms are fairly complicated [18], this may also be interesting for a sequential algorithm. Indeed, a similar approach might also work for other problems where sorting can be a bottleneck, *e.g.,* smoothed analysis of approximate weighting matching [21].

For subset sampling (Problem WRS–S), we parallelize the approach of Bringmann et al. [6] in Section 9. Once more, the preprocessing requires integer sorting. However, only $\mathcal{O}(\log n)$ different keys are needed so that linear work sorting works with logarithmic latency even deterministically on a CREW PRAM.

In Section 10, we adapt the sequential streaming algorithm of Efraimidis et al. [12] to a distributed setting where items are processed in small batches. This can be done in a communication efficient way using our previous work on distributed priority queues [15].

2 Preliminaries

2.1 Models of Computation

We strive to describe our parallel algorithms in a model-agnostic way, *i.e.,* we largely describe them in terms of standard operations such as prefix sums for which efficient parallel algorithms are known on various more models of computation. We analyze the algorithms for two simple models of computation. In each case p denotes the number of processing elements (PEs). Most of our algorithms achieve polylogarithmic running time for a sufficiently large number of PEs. This is a classical goal in parallel algorithm theory and we believe that it is now becoming practically important with the advent of massively parallel ("exascale") computing and fine-grained parallelism in GPGPUs.

For shared-memory algorithms we use the CREW PRAM model (concurrent read exclusive write parallel random access machine) [16]. We will use the concepts of total *work* and *span* of a computation to analyze these algorithms. The span of a computation is the time needed by a parallel algorithm with an unbounded number of PEs.

For distributed-memory computations we use point-to-point communication between PEs where exchanging a message of length ℓ takes time $\alpha + \ell\beta$. We assume $1 \leq \beta \leq \alpha$. We will use that *prefix sums* and *(all-)reductions* can be computed in time $\mathcal{O}(\beta\ell + \alpha \log p)$ for vectors of size ℓ. The *all-gather* operation collects a value from each PE and delivers all values to all PEs. It can be implemented to run in time $\mathcal{O}(\beta p + \alpha \log p)$ [17]. We will particularly strive to obtain *communication efficient algorithms* [30] where total communication cost is sublinear in the local computations. Some of our algorithms are even *communication free.*

We need one basic toolbox operation where the concrete machine model has some impact on the complexity. Sorting n items with integer keys from $1..K$ can be done with linear work in many relevant cases. Sequentially, this is possible if K is polynomial in n (radix sort). Radix sort can be parallelized even on a distributed-memory machine with linear work and span n^ε for any constant $\varepsilon > 0$. Logarithmic span is possible for $K = \mathcal{O}(\log^c n)$ for any constant c, even on an EREW PRAM [26, Lemma 3.1]. For a CRCW PRAM,

expected linear work and logarithmic span can be achieved when $K = \mathcal{O}(n \log^c n)$ [26] (the paper gives the constraint $K = \mathcal{O}(n)$ but the generalization is obvious and important for us in Theorem 14). Resorting to comparison based algorithms, we get work $\mathcal{O}(n \log n)$ and $\mathcal{O}(\log n)$ span on an EREW PRAM [8].

2.2 Bucket-Based Sampling

The basic idea behind several solutions of Problem WRS–1 is to build a table of $m = \Theta(n)$ buckets where each bucket represents a total weight of W/n. Sampling then selects a random bucket uniformly at random and uses the information stored in the bucket to determine the actual item. If item weights differ only by a constant factor, we can simply store one item per bucket and use rejection sampling to obtain constant expected query time (see, *e.g.*, Devroye [9], Olken et al. [25]). Section 4.1 uses $2n$ buckets and rejection sampling to store the items in their given order, using only one bit per bucket.

Deterministic sampling with only a single memory probe is possible using Walker's *(1-)alias table* method [34], and its improved construction due to Vose [33]. Bucket i represents some fraction of the weight of item i (all of it if $w_i \leq W/n$). The remaining weight of the heavier items is distributed to the remaining capacity of the buckets. This is possible in such a way that each bucket only represents one of these large items. Thus, each bucket i stores a weight w'_i (representing a piece of item i) and *one* index a_j of another item (called an *alias*). To sample an item, pick a bucket index r uniformly at random, toss a biased coin that comes up heads with probability $w'_r n/W$, and return r for heads, or a_r for tails. The algorithm for computing the weights and aliases is a simple and elegant linear-time algorithm [33]. However, it seems quite difficult to parallelize it.

2.3 Weighted Sampling using Exponential Variates

It is well known that an (unweighted) sample without replacement of size k out of n items $1..n$ can be obtained by associating with each item a uniform variate $v_i := \texttt{rand}()$, and selecting the k with the smallest associated variates. This method can be generalized to generate a *weighted* sample without replacement by raising uniform variates to the power $1/w_i$ and selecting the k items with the *largest* associated values [10–12]. Equivalently, one can generate exponential random variates $v_i := -\ln(\texttt{rand}())/w_i$ and again select the k items with the *smallest* associated v_i [2] ("*exponential clocks method*"), which is numerically more stable.

2.4 Divide-and-Conquer Sampling

Uniform sampling with and without replacement can be done using a divide-and-conquer algorithm [31]. To sample k out of n items uniformly and with replacement, split the set into two subsets with n' (left) and $n - n'$ (right) items, respectively. Then the number of items k' to be sampled from the left has a binomial distribution (k trials with success probability n'/n). We can generate k' accordingly and then recursively sample k' items from the left and $k - k'$ items from the right. This can be used for a communication-free parallel sampling algorithm. We have a tree with p leaves. Each leaf represents a subproblem of size about n/p – one for each PE. Each PE descends this tree to the leaf assigned to it (time $\mathcal{O}(\log p)$) and then generates the resulting number of samples (time $\mathcal{O}(k/p + \log p)$ with high probability). Different PEs have to draw the same random variates for the same interior node of the tree. This can be achieved by seeding a pseudo-random number generator with an ID of this node.

3 Related Work

3.1 Sampling one Item (Problem WRS–1)

Extensive work has been done on generating discrete random variates from a fixed distribution [6, 9, 19, 33, 34]. All these approaches use preprocessing to construct a data structure that subsequently supports very fast (constant time) sampling of a single item. Bringmann et al. [5] explain how to achieve expected time r using only $\mathcal{O}(n/r)$ bits of space beyond the input distribution itself. There are also dynamic versions that allow efficient weight updates. Some (rather complicated ones) allow that even in constant expected time [14, 20].

3.2 Sampling Without Replacement (Problems WRS–N and WRP)

The exponential clocks method of Section 2.3 is an $\mathcal{O}(n)$ algorithm for sampling without replacement. This approach also lends itself towards use in streaming settings (*reservoir sampling*) and can be combined with a skip value distribution to reduce the number of required random variates from $\mathcal{O}(n)$ to $\mathcal{O}\left(k \log \frac{n}{k}\right)$ [12]. A related algorithm for WRS–N with given inclusion probabilities instead of relative weights is described by Chao [7].

More efficient algorithms for WRS–N repeatedly sample an item and remove it from the distribution using a dynamic data structure [14, 20, 25, 35]. With the most efficient such algorithms [14, 20] we achieve time $\mathcal{O}(k)$, albeit at the price of an inherently sequential and rather complicated algorithm that might have considerable hidden constant factors.

It is also possible to combine techniques for sampling *with* replacement with a rejection method. However, the performance of these methods depends heavily on U, the ratio between the largest and smallest weight in the input, as the rejection probability rises steeply once the heaviest items are removed. Lang [18] gives an analysis and experimental evaluation of such methods for the case of $k = n$ (*cf.* "Permutation" below). A recent practical evaluation of approaches that lend themselves towards efficient implementation is due to Müller [24].

The **Permutation** Problem WRP can be seen as special case of sampling without replacement with $k = n$. In particular, sorting the exponential variates from Section 2.3 computes such a permutation. Lang [18] compares different approaches to Problem WRP, including one based on card shuffling techniques.

3.3 Subset Sampling (Problem WRS–S)

Subset sampling is the generalization of Bernoulli Sampling, which can be solved in optimal expected time linear in the output size by computing the geometrically distributed distances between elements in the sample [1], to unequal probabilities. The naïve algorithm, which consists of throwing a biased coin for each item, requires $\mathcal{O}(n)$ time. Bringmann et al. [6] show that this is optimal if only a single subset is desired, and present a sequential algorithm that is also optimal for multiple queries.

3.4 Parallel Sampling

There is surprisingly little work on parallel sampling. Even uniform unweighted sampling had many loose ends until recently [31]. Parallel uniformly random permutations are covered in [13, 29]. Efraimidis and Spirakis note that WRS–N can be solved in parallel with span $\mathcal{O}(\log n)$ and work $\mathcal{O}(n \log n)$ [11]. They also note that it suffices to solve the selection problem when the output need not be sorted. The optimal dynamic data structure for WRS–

1 [20] admits a parallel bulk update in the (somewhat exotic) combining-CRCW-PRAM model. However, this does not help with Problem WRS–N since their batch sizes are one.

Some results may be viewed as folklore though. Of course, once we have a data structure like the alias table, we can perform sampling with replacement in an embarrassingly parallel way on a shared-memory machine.

4 Parallel Alias Tables (Problem WRS–1)

Alias table construction looks difficult to parallelize because there are complicated dependencies between the computations involving different items. In particular, how much of a large item i is placed into bucket i only becomes apparent as the computation proceeds. In our first data structure, *2-alias tables*, we greatly simplify these dependencies by slightly generalizing the data structure to allow up to *two* aliases per bucket. This increases the space requirements by a constant factor and slightly increases the query time (around 20 % in our experiments, see Section 11) but allows for fast parallel construction. For algorithms with less space consumption, refer to the distributed algorithm in Section 5 and to the compressed data structure outlined in Section 4.1.

Table Construction

As before, we have an array of n buckets with capacity W/n and we distinguish between small items ($w_i \leq W/n$) and large items ($w_i > W/n$). Once more, small items are completely assigned to one bucket and the large items are distributed over the remaining capacity r_i of the buckets. To do that in parallel, we compute a prefix sum $F_i := \sum_{j \leq i} r_j$ over these remaining capacities as well as a prefix sum L_j over the sizes of the large items. Now, we assign a range of $\underline{i}..\bar{i}$ of n/p buckets to each PE which is responsible for filling the remaining capacity of these buckets with pieces of large items. These pieces can be identified by binary search in L and stem from a range of at most n/p large items. More specifically, let j denote the smallest position in L such that $L_j > F_{\underline{i}-1}$. Then the j-th large item is the first one of which a piece has to be placed in buckets $\underline{i}..n$. This piece has size $L_j - F_{\underline{i}-1}$. This large item (and possibly subsequent ones) are placed into consecutive buckets until bucket $\bar{i}$ is completely filled. A remaining piece of a partially placed item will be handled as the starting piece of the next PE. All PEs simultaneously scan their range of buckets and large items. In each step, a PE exhausts either a bucket or a large item. In particular, each bucket is assigned at most two pieces of different large items because no large item entirely fits into any bucket. An example is shown in Figure 1.

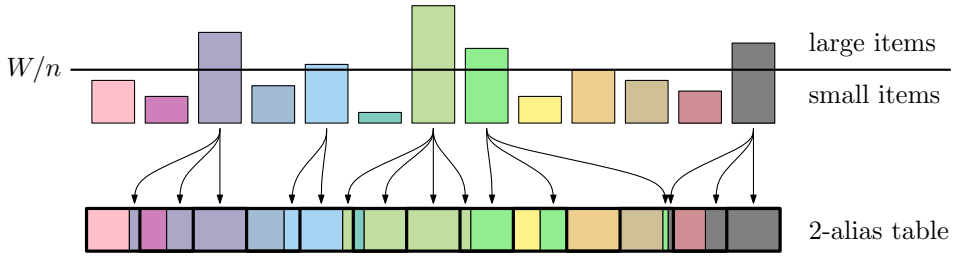


Figure 1 An example of a 2-alias table with $n = 13$, $W/n = 10$. Items are represented by colors. Assignment of large items is illustrated with arrows.

Sampling from the Resulting Table

For sampling a single item, generate a uniform random variate $u \in [0,1)$ to choose a bucket $b = \lfloor un \rfloor$. Then the fractional part $r := un - b$ is a uniform random variate from $[0,1)$.[1] Suppose bucket b contains a small item i (a zero size dummy item if it contains no small item), a piece of a large item j of size s_j, and possibly a piece of a second large item k. If $r < w_i$ return i. Otherwise, if $r < w_i + s_j$, return j. Return k in the remaining cases. Note that the size of the piece from item k is not needed here.

▶ **Theorem 1.** *We can construct a 2-alias table with work $\Theta(n)$ and span $\Theta(\log n)$ on a CREW PRAM. The resulting data structure supports sampling a single item in $\Theta(1)$ time by a single PE.*

Proof. Computing prefix sums is a standard operation that has the desired work and span [4]. Binary search takes time $\mathcal{O}(\log n)$ on each each PE. Each PE scans n/p buckets and $\leq n/p$ large items. ◀

4.1 Compressed data structures for WRS–1

Bringmann and Larsen [5] give a construction similar to alias tables that allows expected query time $\mathcal{O}(r)$ using $2n/r + o(n)$ bits of additional space. We describe the variant for $r = 1$ in some more detail. We assign $\lceil w_i/W \rceil$ buckets to each item, *i.e.*, $\leq 2n$ in total. Item i is assigned to buckets $\sum_{j<i} \lceil w_j/W \rceil .. \sum_{j \leq i} \lceil w_j/W \rceil - 1$. A query samples a bucket j uniformly at random. Suppose bucket j is assigned to item i. If $j \in \sum_{j<i} \lceil w_j/W \rceil .. \sum_{j \leq i} \lceil w_j/W \rceil - 2$, item i is returned. If $j = \sum_{j \leq i} \lceil w_k/W \rceil - 1$, item i is returned with probability $\lceil w_i/W \rceil - \lfloor w_i/W \rfloor$. Otherwise, bucket j is rejected and the query starts over. Since the overall success probability is $\geq 1/2$, the expected query time is constant. Even without data compression, the simplicity of this approach will be useful in Section 5.

The central observation for compression is that it suffices to store one bit for each bucket that indicates whether a new item starts at bucket $b[i]$. When querying bucket j, the item stored in it can be determined by counting the number of 1-bits up to position j. This *rank*-operation can be supported in constant time using an additional data structure with $o(n)$ bits. Further reduction in space is possible by representing r items together as one entry in b.

Both constructing the bit vector and constructing the rank data structure is easy to parallelize using prefix sums (for adding scaled weights and counting bits, respectively) and embarrassingly parallel computations. Shun [32] even gives a bit parallel algorithm needing only $\mathcal{O}(n/\log n)$ work for computing the rank data structure. We get the following result:

▶ **Theorem 2.** *Bringmann and Larsen's $n/r + o(n)$ bit data structure can be built using $\mathcal{O}(n)$ work and $\mathcal{O}(\log n)$ span allowing queries in expected time $\mathcal{O}(r)$.*

5 Distributed-Memory Alias Tables

The parallel algorithm described in Section 4 can also be adapted to a distributed-memory machine. However, this requires information about all items to be communicated. Hence, more communication efficient algorithms are important for large n. To remedy this problem,

[1] There is a variant of the data structure for integer weights where a single uniform variate from $0..W-1$ is used to address $\lceil W/B \rceil = \Theta(n)$ buckets of size $B = \lfloor W/n \rfloor$ (or, more efficiently, $B = 2^{\lfloor \log W/n \rfloor}$).

we will now view sampling as a 2-level process implementing the owner-computes approach underlying many distributed algorithms.

Let E_i denote the set of items allocated to PE i. For each PE i, we create a *meta-item* of weight $W_i := \sum_{j \in E_i} w_j$. Sampling now amounts to sampling a meta-item and then delegating the task to sample an actual item from E_i to PE i. The local data structures can be built independently on each PE.[2] In addition, we need to build a data structure for sampling a meta-item. There are several variants in this respect with different trade-offs:

▶ **Theorem 3.** *Assuming that $\mathcal{O}(n/p)$ elements are allocated to each PE, we can sample a single item in time $\mathcal{O}(\alpha)$ after preprocessing a 2-level alias table, which can be done in time $\mathcal{O}(n/p)$ plus the following communication overhead*

$$\beta p + \alpha \log p \text{ with replicated preprocessing} \tag{1}$$

$$\alpha \log^2 p \text{ expected time using the algorithm from Section 4} \tag{2}$$

$$\alpha \log p \text{ with only expected time bounds for the query} \tag{3}$$

Proof. Building the local 1-alias tables takes time $\mathcal{O}(\max_i |E_i|) = \mathcal{O}(n/p)$ sequentially. For Equation (1), we can perform an all-gather operation on the meta-items and compute the data structure for the meta-items in a replicated way.

For Equation (2) and Equation (3), we can compute an alias table for the meta-items using a parallel algorithm. Sampling then needs an additional indirection. First, a meta-bucket j is computed. Then a request is sent to PE j which identifies the subset E_i from which the item should be selected and delegates that task of sampling from E_i to PE i.[3] Equation (2) then follows by using the shared-memory algorithm from Section 4. It can be implemented to run in expected time $\mathcal{O}(\alpha \log^2 p)$ on a distributed-memory machine using PRAM emulation [27].

At the price of getting only expected query time, we can also achieve logarithmic latency (Equation (3)) by using the rejection sampling algorithm from Section 4.1. The preprocessing there requires only prefix sums that can directly be implemented on distributed memory: We have to assign p meta-items to $2p$ meta-buckets (two on each PE). Suppose PE i computes the prefix sum $k = \sum_{j<i} \lceil W_j/W \rceil$. It then sends item i to PE $j = \lfloor k/2 \rfloor$. PE j then initiates a broadcast of item i to PEs $j..\lfloor (j + \lfloor W_i/W \rfloor - 1)/2 \rfloor$. All of this is possible in time $\mathcal{O}(\alpha \log p)$. ◀

5.1 Redistributing Items

As discussed so far, *constructing* distributed-memory 2-level alias tables is communication efficient. However, when large items are predominantly allocated on few PEs, *sampling* many items can lead to an overload on PEs with large W_i. We can remedy this problem by moving large items to different PEs or even by splitting them between multiple PEs. This redistribution can be done in the same way we construct 2-alias tables. This implies a trade-off between redistribution cost (part of preprocessing) and load balance during sampling.

▶ **Theorem 4.** *If items are randomly distributed over the PEs initially, it suffices to redistribute $\mathcal{O}(\log p)$ items from each PE such that afterwards each PE has total weight $\mathcal{O}(W/p)$ in expectation and $\mathcal{O}(n/p + \log p)$ (pieces of) items. This redistribution takes expected*

[2] Possibly using a shared-memory parallel algorithm locally.

[3] If we ensure that meta-items have similar size (see Section 5.1) then we can arrange the meta-items in such a way that $i = j$ most of the time.

time $\mathcal{O}(\alpha \log^2 p)$ *when supporting deterministic queries (Theorem 3-(2)) and expected time* $\mathcal{O}(\alpha \log p)$ *using rejection sampling (Theorem 3-(3)).*

Proof. Let us distinguish between *heavy* items that are larger than $cW/(p \log p)$ for an appropriate constant c and the remaining *light* items. The expected maximum weight allocated to a PE based on light items is $\mathcal{O}(W/p)$ [28].

There can be at most $p \log(p)/c$ heavy items. By standard balls into bins arguments, only $\mathcal{O}(\log p)$ heavy items can initially be allocated to any PE with high probability. We use the algorithm from Theorem 3-(2) to distribute the heavy items to p meta-buckets of remaining capacity $\max(0, W/p - S_i)$ where S_i is the total weight of the light items allocated to PE i. Using the bound from Equation (2) would result in a time bound of $\mathcal{O}(\log^3 p)$ since we have a factor $\mathcal{O}(\log p)$ more items. However, the only place where we need a full-fledged PRAM emulation is for doing the binary search which takes only $\mathcal{O}(\log p)$ steps on the PRAM and time $\mathcal{O}(\alpha \log^2 p)$ when emulated on distributed memory.

For the faster variant with rejection sampling, we use prefix sums to distribute the largest $N := \mathcal{O}(p \log p)$ items such that each PE gets an even share of it. For this, we build groups of $N/p = \mathcal{O}(\log p)$ items that we distribute in an analogous fashion to the proof in Theorem 3-(3) – a prefix sum, followed by forwarding a group followed by a segmented broadcast. The asymptotic complexity does not change since even messages of size $\mathcal{O}(\log p)$ can be broadcast in time $\mathcal{O}(\alpha \log p)$, *e.g.,* using pipelining. Finally, each PE unpacks the group it received and extracts the parts that it has to represent in the meta-table. ◀

6 Output Sensitive Sampling With Replacement (Problem WRS–R)

The algorithm of Section 4 easily generalizes to sampling k items with replacement by simply executing k queries. Since the precomputed data structures are immutable, these queries can be run in parallel. We obtain optimal span $\mathcal{O}(1)$ and work $\mathcal{O}(k)$.

▶ **Corollary 5.** *After a suitable alias table data structure has been computed, we can sample* k *items with replacement with work* $\mathcal{O}(k)$ *and span* $\mathcal{O}(1)$*.*

Yet if the weights are skewed this may not be optimal since large items will be sampled multiple times. Here, we describe an *output sensitive* algorithm that outputs only different items in the sample together with how often they were sampled, *i.e.,* a set S of pairs (i, k_i) indicating that item i was sampled k_i times. The work will be proportional to the output size s up to a small additive term. For highly skewed distributions, even $k \gg n$ may make sense.

Note that outputting multiplicities may be important for appropriately processing the samples. For example, let X denote a random variable where item i is sampled with probability w_i/W and suppose we want a truthful estimator for the expectation of $f(X)$ for some function f. Then $\sum_{(i,k_i)\in S} k_i f(i)/k$ is such an estimator.

We will combine and adapt three previously used techniques for related problems: the bucket tables from Section 2.2, the divide-and-conquer technique from Section 2.4 [31], and the subset sampling algorithm of Bringmann et al. [6].

We approximately sort the items into $u = \lceil \log U \rceil$ groups of items whose weight differs by at most a factor of two – weight w_i is mapped to group $\lfloor \log(w_i/w_{\min}) \rfloor$. In contrast to subset sampling, this has to extend to items with even the smallest weights.

To help determine the number of samples to be drawn from each group, we build a complete binary tree with one *nonempty* group at each leaf. Interior nodes store the total weight of items in groups assigned to their subtrees. This *divide-and-conquer tree (DC-tree)*

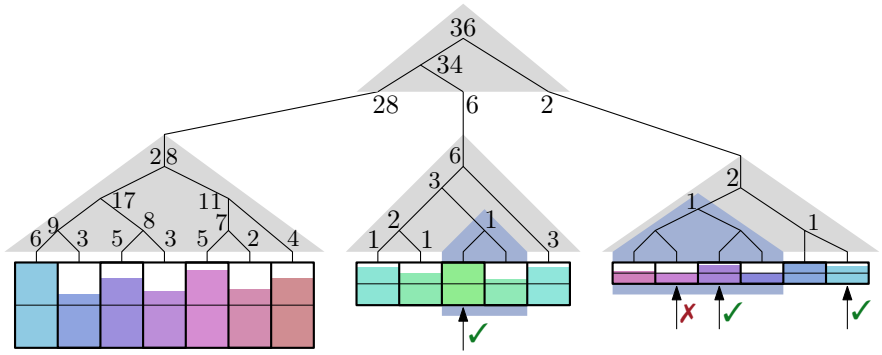


Figure 2 Output-sensitive sampling: assignment of multiplicities with $k = 36$.

allows us to generalize the technique from Section 2.4 to weighted items. Suppose we want to sample k elements from a subtree rooted at an interior node whose left subtree has total weight L and whose right subtree has total weight R. Then the number of items k' to be sampled from the left has a binomial distribution (k trials with success probability $L/(L+R)$). We can generate k' accordingly and then recursively sample k' items from the left subtree and $k - k'$ items from the right subtree. A recursive algorithm can thus split the number of items to be sampled at the root into numbers of items sampled at each group. When a subtree receives no samples, the recursion can be stopped. Since the distribution of weights to groups can be highly skewed, this stopping rule will be important in the analysis.

For each group G, we integrate bucket tables and DC-tree as follows. For the bucket table we can use a very simple variant that stores n_G items with weights from the interval $[a, 2a)$ in n_G buckets of capacity $2a$. Sampling one element then uses a rejection method that repeats the sampling attempt when the random variate leads to an empty part of a bucket.[4]

We also build a DC-tree for each group. A simple linear mapping of items into the bucket table allows us to associate a range of relevant buckets b_T with each subtree T of the DC-tree.

For sampling m items from a group G, we use the DC-tree to decide which subtree has to contribute how many samples. When this number decreases to 1 for a subtree T, we sample this element directly and in constant expected time from the buckets in the range b_T.

Figure 2 gives an example. We obtain the following complexities:

▶ **Theorem 6.** *Preprocessing for Problem WRS–R can be done in the time and span needed for integer sorting n elements with $u = \lceil \log U \rceil = \lceil \log(w_{\max}/w_{\min}) \rceil$ different keys[5] plus linear work and logarithmic span (on a CREW PRAM) for building the actual data structure. Using this data structure, sampling a multiset S with k items and s different items can be done with span $\mathcal{O}(\log n)$ and expected work $\mathcal{O}(s + \log n)$ on a CREW PRAM.*

Proof. Besides sorting the items into groups, we have to build binary trees of total size $\mathcal{O}(n)$. This can be done with logarithmic span and linear work using a simple bottom-up reduction. The bucket-tables which have total size n can be constructed as in Section 4.

The span of a query is essentially the depth of the trees, $\log u + \log n \le 2 \log n$.

Bounding the work for a query is more complicated since, in the worst case, the algorithm can traverse paths of logarithmic length in the DC-trees for just a constant number of samples

[4] If desired, we can also avoid rejection sampling by mapping the items without gaps into buckets of size $\sum_{i \in G} w_i / n_G$. This way there are at most three items in each bucket.

[5] Section 2.1 discusses the cost of this operation on different models of computation.

taken at its leaves. However, this is unlikely to happen and we show that in expectation the overhead is a constant factor plus an additive logarithmic term. We are thus allowed to charge a constant amount of work to each different item in the output and can afford a leftover logarithmic term.

We first consider the top-most DC-tree T that divides samples between groups. Tree T is naturally split into a *heavy* range of groups that contain some items which are sampled with probability at least $1/2$ and a remaining range of *light* groups in which all items are sampled with probability at most $1/2$. Assuming the heavy groups are to the left, consider the path P in T leading to the first light group. Subtrees branching from P to the left are complete subtrees that lead to heavy groups only. Since all leaves represent non-empty groups, we can charge the cost for traversing the left trees to the elements in the groups at the leaves – in expectation, at least half of these groups contain elements that are actually sampled.

Then follow at most $2 \log n$ light groups that have a probability $\geq 1/n$ to yield at least one sample. These middle groups fit into subtrees of T of logarithmic total size and hence cause logarithmic work for traversing them.

The expected work for the remaining very light groups can be bounded by their number ($\leq u \leq n$) times the length of the path in T leading to them ($\leq \log u \leq \log n$) times the probability that they yield at least one sample ($\leq 1/n$). The product ($\leq n \log(n)/n = \log n$) is another logarithmic term.

Finally, Lemma 7 shows that the work for traversing DC-trees within a group is linear in the output size from each group. Summing this over all groups yields the desired bound. ◀

▶ **Lemma 7.** *Consider a DC-tree plus bucket array for sampling with replacement of k out of n items where weights are in the range $[a, 2a)$. Then the expected work for sampling is $\mathcal{O}(s)$ where s is the number of different sampled items.*

Proof. If $k \geq n$, $\Omega(n)$ items are sampled in expectation at a total cost of $\mathcal{O}(n)$. So assume $k < n$ from now on. The first $\log k + \mathcal{O}(1)$ levels of T may be traversed completely, contributing a total cost of $\mathcal{O}(k)$.

For the lower levels, we count the number Y of visited nodes from which at least 2 items are sampled. This is proportional to the total number of visited nodes since nodes from which only one item is sampled contribute only constant expected cost (for directly sampling from the array) and since there are at most $2Y$ such nodes.

Let X denote the number of items sampled at a node at level ℓ of tree T. An interior node at level ℓ represents $2^{L-\ell}$ leaves with total weight $W_\ell \leq 2a2^{L-\ell}$ where $L = \lceil \log n \rceil$. X has a binomial distribution with k trials and success probability

$$p = \frac{W_\ell}{W} \leq \frac{2a2^{L-\ell}}{a2^{L-1}} = 4 \cdot 2^{-\ell} \ .$$

Hence,

$$\mathbf{P}[X \geq 2] = 1 - \mathbf{P}[X = 0] - \mathbf{P}[X = 1] = 1 - (1-p)^k - kp(1-p)^{k-1} \approx (kp)^2/2$$

where the "$\approx$" holds for $kp \ll 1$ and was obtained by series development in the variable kp.

The expected cost at level $\ell > \log k + \mathcal{O}(1)$ is thus estimated as

$$2^l \mathbf{P}[X \geq 2] \approx 2^l (kp)^2/2 \leq 2^\ell (k \cdot 4 \cdot 2^{-\ell})^2/2 = 8k^2 2^{-\ell} \ .$$

At level $\ell = \lceil \log k \rceil + 3 + i$ we thus get expected cost $\leq k2^{-i}$. Summing this over i yields total cost $Y = \mathcal{O}(k)$. ◀

6.1 Distributed Case

The batched character of sampling with replacement makes this setting even more adequate for a distributed implementation using the owner-computes approach. Each PE builds the data structure described above for its local items. Furthermore, we build a top-level DC-tree that distributes the samples between the PEs, *i.e.,* with one leaf for each PE. We will see below that this can be done using a bottom-up reduction over the total item weights on each PE, *i.e.,* no PRAM emulation or replication is needed. Each PE only needs to store the partial sums appearing on the path in the reduction tree leading to its leaf. Sampling itself can then proceed without communication – each PE simply descends its path in the top-level DC-tree analogous to the uniform case [31]. Afterwards, each PE knows how many samples to take from its local items. Note that we assume k to be known on all PEs and that communication for computing results from the sample is not considered here.

▶ **Theorem 8.** *Sampling k out of n items with replacement (Problem WRS–R) can be done in a communication-free way with processing overhead $\mathcal{O}(\log p)$ in addition to the time needed for taking the local sample. Building and distributing the DC-tree for distributing the samples is possible in time $\mathcal{O}(\alpha \log p)$.*

Proof. It remains to explain how the reduction can be done in such a way that it can be used as a DC-tree during a query and such that each PE knows the path in the reduction tree leading to its leaf. First assume that $p = 2^d$ is a power of two. Then we can use the well known hypercube algorithm for all-reduce (*e.g.,* [17]). In iteration $i \in 1..d$ of this algorithm, a PE knows the sum for its local $i-1$ dimensional subcube and receives the sum for the neighboring subcube along dimension i to compute the sum for its local i dimensional subcube. For building the DC-tree, each PE simply records all these values.

For general values of p, we first build the DC tree for $d = \lfloor \log p \rfloor$. Then, each PE i with $i < 2^d$ and $j = i + 2^d < p$ receives the aggregate local item weight from PE j and then sends its path to PE j. ◀

Similar to Section 5, it depends on the assignment of the items to the PEs whether this approach is load balanced for the local computations. Before, we needed a balanced distribution of both number of items and item weights. Now the situation is better because items may be sampled multiple times but require work only once. On the other hand, we do not want to split heavy items between multiple PEs since this would increase the amount of work needed to process the sample. It would also undermine the idea of communication-free sampling if we had to collect samples of the same item assigned to different PEs. Below, we once more analyze the situation for items with arbitrary weight that are allocated to the PEs randomly.

▶ **Theorem 9.** *Consider an arbitrary set of item sizes and let $u = \log(\max_i w_i / \min_i w_i)$. If items are randomly assigned to the PEs initially, then preprocessing takes expected time $\mathcal{O}\big(\text{isort}_u^1(n/p) + \alpha \log p\big)$ where $\text{isort}_u^1(x)$ denotes the time for sequential integer sorting of x elements using keys from the range $0..u$.[6] Using this data structure, sampling a multiset S with k items and s different items can be done in expected time $\mathcal{O}(s/p + \log p)$.*

Proof. For preprocessing, standard Chernoff bound arguments tell us that $\mathcal{O}(n/p + \log p)$ items will be assigned to a PE with high probability. Since sorting is now a local operation,

[6] Note that this will be linear in all practically relevant situations.

we only need an efficient sequential algorithm for approximately sorting integers. The term $\alpha \log p$ is for the global DC-tree as in Theorem 8.

A sampling operation will sample s items. Since their allocation is independent of the choice of the sampled items, we can once more use Chernoff bounds to conclude that only $\mathcal{O}(s/p + \log p)$ of them are allocated to any PE with high probability. ◀

7 Sampling k Items Without Replacement (Problem WRS–N)

We can construct an algorithm for sampling without replacement based on the algorithm for sampling *with* replacement of Section 6. Presume we know an $\ell > k$ so that a sample with replacement contains at least k and no more than $\mathcal{O}(k)$ unique items. Then we can obtain such a sample with multiplicities of size $k' \geq k$ using the algorithm of Section 6, discard the multiplicities, and downsample to size k using the exponential clocks method (see Section 2.3).

The crucial step, of course, is to find ℓ. This depends heavily on the distribution of the input: if all items have similar weight, ℓ will be little larger than k, but if the weights are heavily skewed, it could have to be exponentially large. We can bound the number of unique items in a sample with replacement of size ℓ by considering the probability of any item to *not* be sampled. Item i is *not* part of the sample with probability $(1 - w_i/W)^\ell$, and we obtain $\mathbf{E}[X] = \sum_{i=1}^{n} 1 - (1 - w_i/W)^\ell$ where X is the random variable describing the number of unique items in the sample.

▶ **Lemma 10.** $\left(1 - \frac{1}{e}\right) t_\ell \leq \mathbf{E}[X] \leq t_\ell$ *where* $t_\ell := \ell \cdot \sum_{i:\, w_i < \frac{W}{\ell}} \frac{w_i}{W} + \left|\left\{ i \;\middle|\; \frac{w_i}{W} \geq \frac{1}{\ell} \right\}\right|$.

Proof. We split the formula for the expectation of X into two parts: the items for which $w_i/W < 1/\ell$, and those with $w_i/W \geq 1/\ell$. For the first group, by Bernoulli's inequality, we obtain $1 - (1 - w_i/W)^\ell \leq \ell w_i/W$. Furthermore, we have:

$$\begin{aligned} 1 - (1 - w_i/W)^\ell &= 1 - (1 - b_i/n)^{c \cdot n} && \text{where } b_i := n w_i/W \text{ and } c := \ell/n \\ &\geq 1 - e^{-c b_i} = e^{-\ell w_i/W} && \text{by } e^x \geq (1 + x/n)^n \\ &\geq (1 - 1/e)\, \ell w_i/W && \text{because } e^{-x} \leq 1 - (1 - 1/e)x \text{ for } x \in (0, 1) \end{aligned}$$

and thus the first term of t is explained. Consider now the items with $w_i/W \geq 1/\ell$, *i.e.,* the items which yield at least one sample in expectation. Clearly, $1 - (1 - w_i/W)^\ell \leq 1$. In the other direction, because $w_i/W \geq 1/\ell$, we obtain $1 - (1 - w_i/W)^\ell \geq 1 - (1 - 1/\ell)^\ell \geq 1 - 1/e$ by the inequality $(1 + x/n)^n \leq e^x$ for $\ell > 1$. Otherwise, if $\ell \leq 1$, either no item with $w_i/W \geq 1/l = 1$ exists, or we are in the trivial case with $n = 1$ and $w_1 = W$. Summing the results for the first and second group, we obtain the claimed inequalities. ◀

An example of this is illustrated in Figure 3. By applying the above estimation to the groups used by the algorithm of Section 6, we can quickly obtain an estimate for the output size that is at most a factor of two worse.

▶ **Lemma 11.** *Applying the estimation of Lemma 10 to groups of items of similar weight as in Section 6 loosens the bound on* $\mathbf{E}[X]$ *by at most a factor of two.*

Proof. Item i is in group $\lfloor \log(w_i/w_{\min}) \rfloor$. Label the groups $G_1..G_u$ where $u := \lceil \log U \rceil = \lceil \log(w_{\max}/w_{\min}) \rceil$ and let the value range of group i be $[a_i, 2a_i)$. The difference to Lemma 10 now is that we can only choose ℓ as $\lceil 1/(2a_i) \rceil$ for some group i. This necessitates moving entire

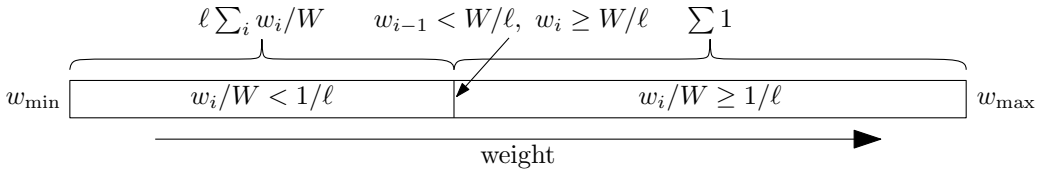


Figure 3 How the expected number of samples $\mathbf{E}[X]$ is influenced by the choice of ℓ depends on the distribution of the inputs (Lemma 10). Items with small probability of being sampled contribute linearly in their weight, whereas large items are expected to be sampled.

groups between the left and right halves of the estimation. Pick a group i, *i.e.*, $\ell := \lceil 1/(2a_i) \rceil$, and consider the effects of choosing group $i+1$ instead, *i.e.*, $\ell' := \lceil 1/(2a_{i+1}) \rceil = \lceil 1/(4a_i) \rceil = \ell/2$. Clearly, the contribution of any group $j > i+1$ is the same for t_ℓ and $t_{\ell'}$. Furthermore, the contribution of groups $1..i$ is halved, as $\ell' = \ell/2$. It remains to bound the contribution of group $i+1$, which is moved from the "right" part of the estimate to the "left" part: it contributes $|G_{i+1}|$ to t_ℓ, and $\xi := \ell' \sum_{j \in G_{i+1}} w_j/W$ to $t_{\ell'}$. However, as all items in G_{i+1} are in the range $[2a_i, 4a_i)$ and $\ell' = \lceil 1/(4a_i) \rceil$, we have $\xi \geq \ell' 2a_i \, |G_{i+1}| \geq 2a_i/(2a_{i+1}) \cdot |G_{i+1}| = |G_{i+1}|/2$, and thus the contribution of group $i+1$ at most doubles, too. Thus, $t_\ell \leq 2t_{\ell'}$, and for any ℓ of Lemma 10 we can find an $\hat{\ell}$ that yields at most twice as many unique items in expectation while only relying on information about the groups, not the w_i. ◀

▶ **Theorem 12.** *Preprocessing for Problem WRS–N is possible with the same work and time bounds as for Problem WRS–R (Theorem 6). Sampling k items without replacement is then possible with span $\mathcal{O}(\log n)$ and expected work $\mathcal{O}(k)$ on a CREW PRAM.*

Proof. First, compute W and apply the preprocessing of Section 6. Then, compute each group's relative total weight, $g_i := \sum_{j \in G_i} w_j/W$, where G_i denotes the group with index i, and their prefix sum, $h_i := \sum_{j \leq i} g_i$. Further, let $c_i := \sum_{j \leq i} |G_j|$ be the number of items in all groups up to group i. All of these steps are possible with linear work and logarithmic span on top of the preprocessing for WRS–R of Theorem 6.

At query time, we can find a suitable value of ℓ in time $\mathcal{O}(\log n)$ using binary search over the non-empty groups. These are exactly the leaves of the top-level DC-tree constructed by the preprocessing for WRS–R. Let i be the index of the group currently under consideration and $[a_i, 2a_i)$ be the interval of probabilities assigned to the group. Then we can evaluate t_ℓ for $\ell := \lceil 1/(2a_i) \rceil$ in constant time as $t_\ell = \ell \cdot h_i + (n - c_i)$. Lemma 11 states that we can find a group i whose maximum weight $2a_i$ gives us a value of ℓ such that sampling ℓ elements with replacement yields $2k$ unique elements in expectation, but at the same time not too many more ($\mathcal{O}(k)$). This ensures that the output contains at least k unique elements with sufficient probability. However, this group may be empty and thus not considered in the binary search. Thus, if the binary search ended between two adjacent *non-empty* groups G_i and G_j, we can solve the linear equation of Lemma 10 for ℓ in constant time.

We then draw a sample *with* replacement of size ℓ as in Theorem 6, discarding the multiplicities. If the resulting sample has fewer than k unique elements, it is rejected and the sampling is repeated. By Markov's inequality, the probability of this happening is at most $1/2$, as ℓ was chosen to yield $\geq 2k$ samples in expectation. In a postprocessing step, we downsample the resulting sample with replacement to the required size k using the exponential clocks technique of Section 3.2. This needs $\mathcal{O}(k)$ work and $\mathcal{O}(\log n)$ span for a selection of k out of $\Theta(k)$ items [15, Section 4]. ◀

7.1 Distributed Case

We again use the owner-computes approach and adapt the distributed data structure for Problem WRS–R from Section 6.1. However, to find the right estimate for the number ℓ of samples with replacement, we need to perform a global estimation taking all items into account. This can be achieved by finding the global sum of all the local prefix sums in each step of the binary search. This increases the latency of the algorithm to $\mathcal{O}(\alpha \log(p) \log(n))$. Also, in each iteration we need a nested binary search in order to find corresponding buckets in the local bucket arrays.

If the global number of buckets u is not too large, we can consider a different trade-off between preprocessing cost and query cost. We precompute a replicated array of global bucket sizes which allows finding the right value of ℓ with local work $\mathcal{O}(\log u) \leq \mathcal{O}(\log n)$. Local sorting is then performed using bucket sort with u buckets in time $\mathcal{O}(n/p + u)$. The global bucket sizes can then be computed as an all-reduction on the local arrays of bucket sizes in time $\mathcal{O}(\beta u + \alpha \log p)$.

During a query, after sampling with replacement, we need a global parallel selection algorithm to reduce the sample size to k. This can be done in expected time $\mathcal{O}(\beta k/p + \alpha \log p)$ using the unsorted selection algorithm from [15, Section 4]. The term $\beta k/p$ stems from the need to redistribute samples taken within the selection algorithm. For randomly distributed data this is not necessary and the term becomes a local work term k/p. We get the following result:

▶ **Theorem 13.** *Consider an arbitrary set of item sizes and let* $u = \log(\max_i w_i / \min_i w_i)$. *If items are randomly assigned to the PEs initially, then preprocessing takes expected time* $\mathcal{O}\big(\text{isort}^1_u(n/p) + \alpha \log p\big)$ *where* $\text{isort}^1_u(n/p)$ *denotes the time for sequential integer sorting of* x *elements using keys from the range* $0..u$. *Using this data structure, sampling* k *items without replacement can be done in expected time* $\mathcal{O}(k/p + \alpha \log(n) \log(p))$.

A variant that uses a replicated array of global bucket sizes works with preprocessing time $\mathcal{O}(n/p + \beta u + \alpha \log p)$ *and query time* $\mathcal{O}(k/p + \log u + \alpha \log p)$.

8 Permutation (Problem WRP)

As already explained in Section 2.3, weighted permutation can be reduced to sorting random variates of the form $-\ln(r)/w_i$ where r is a uniform random variate. The nice thing is that we know a lot about parallel sorting. The down-side is that sorting may need superlinear work in the worst case. However, since we are sorting *random* numbers, we may still get linear expected work. This is well known when sorting uniform random variates; *e.g.,* [22, Theorem 5.9]. The idea is to map the random variates in linear time to a small number of buckets such that the occupancy of a bucket is bounded by a binomial distribution with constant expectation. Then the buckets can be sorted using a comparison based algorithm without getting more than linear work in total. Here we explain how to achieve the same for the highly skewed distribution needed for WRP.

We use the monotonous transformation of the above mapping function to $f(r, w_i) = n \ln(-\ln(r) n w_{\max}/w_i)$ where $w_{\max} = \max_j w_j$. Except for an expected constant number of elements, r will be in the range $[1/n, 1 - 1/n]$. This corresponds to a range of

$$\left[n \ln -\frac{\ln(1-1/n) n w_{\max}}{w_{\max}}, n \ln -\frac{\ln(1/n) n w_{\max}}{w_{\min}} \right] \approx [0, n \ln(nU \ln n)]$$

for f where $w_{\min} = \min_j w_j$ and $U = w_{\max}/w_{\min}$. Values outside this range will be mapped to keys -1 and $\lceil n \ln(nU \ln n) \rceil$. The remaining items will be mapped to the integer key

$\lfloor f(r, w_i) \rfloor$. We perform integer sorting on the truncated keys and then apply comparison based sorting to the resulting buckets with the same integer keys.

▶ **Theorem 14.** *Problem WRP can be solved in the work and span needed for integer sorting of n items with keys from the range $0..\mathcal{O}(n \ln(nU))$.*

Proof. The main proof obligation is to show that buckets have constant expected occupancy regardless of the weight distribution. First note that f can be written as $n(\ln(-\ln r) + \ln(nw_{\max}/w_i))$. The factor n just scales the bucket size. The two terms multiplied with n separate the influence of the uniform variate r and of the weights. This means that the weight term just shifts the distribution produced by the term $\ln(-\ln(r))$, *i.e.,* the overall distribution is a mix of shifted versions of the same distribution – one for each weight value. This means that the maximum bucket occupancy is maximized if all weights are the same. Let us ignore the scaling factor n and the shift for now and concentrate on the mapping $g(r) = \ln(-\ln r)$. The value of r needed to produce a value x is e^{-e^x}. The value of r needed to produce a value $x + \delta$ is $e^{-e^{x+\delta}}$. Hence, the probability to produce a value in $[x, x + \delta]$ is $e^{-e^x} - e^{-e^{x+\delta}}$. In other words, the probability density is maximized at x when the derivative of e^{-e^x} is minimized. Using calculus, it can be shown that this is the case at $x = 0$. Hence, the probability that an item is mapped to a bucket is bounded by the width of the bucket times $e^{-e^0} = 1/e$. Taking the scaling factor n into account now, we see that the bucket width is $1/n$, yielding a probability of $\leq 1/(ne)$ for any item to be mapped to any particular bucket. Hence, the occupancy of any bucket is bounded by a binomial distribution with success probability $\leq 1/(ne)$ and n trials, *i.e.,* with an expected occupancy of $1/e$.

The proof from [22] for the uniform case transfers for the cost of sorting the buckets with overall work $\mathcal{O}(n)$. The span for that part is $\mathcal{O}(\log n)$ with high probability. This can easily be shown using Chernoff bounds. ◀

Depending on U and the machine model, this can yield very efficient algorithms. When span n^ϵ for some constant ϵ is acceptable, we get linear work if $\log U$ is polynomial in n using radix sort. If U itself is polynomial in n we get logarithmic span on a CRCW PRAM [26].

9 Subset Sampling (Problem WRS–S)

If we do not have a fixed sample size but rather treat the items' weights as independent inclusion probabilities in the sample (this requires $w_i \leq 1$ for all i), different algorithms are required. Observe that the expected sample size is $W \leq n$. Then our goal is to devise a parallel preprocessing algorithm with work $\mathcal{O}(n)$ which subsequently permits sampling with work $\mathcal{O}(1 + W)$.

We now parallelize the approach of Bringmann et al. [6].

▶ **Theorem 15.** *Preprocessing for Problem WRS–S can be done with work $\mathcal{O}(n)$ and span $\mathcal{O}(\log n)$. A query can then be implemented with expected $\mathcal{O}(W + 1)$ work and $\mathcal{O}(\log n)$ span.*

Proof. Approximately sort the items into $L + 1$ buckets with $L := \lceil \log n \rceil$, where bucket i is $B_i := \{j \mid 2^{-i} \geq w_j \geq 2^{-(i+1)}\}$ for $i \in 0..L-1$ and $B_L := \{j \mid 2^{-L} \geq w_j\}$ contains all sufficiently improbable elements. This can be done with linear work and logarithmic span on a CREW PRAM [26, Lemma 3.1]. Let σ denote the permutation of the items implied by this re-ordering.

Next, we pre-compute an assignment of consecutive ranges of permuted items to PEs. Observe that we need not care about bucket boundaries regarding the assignment; there are no dependencies between elements, and the result remains a valid subset sample. Therefore,

we compute a prefix sum $\hat{w}_i := \sum_{j \le i} w_{\sigma(j)}$ over the inclusion probabilities in their new order. PE i then handles the items whose $\hat{w}_j$ fall into the range $[(i-1)W/p,\ i \cdot W/p)$. These can be found with linear work and constant span by checking for every neighboring pair of items whether a boundary falls between them, and if so, which PE's.

Sampling then proceeds on all buckets in parallel using the assignment calculated during preprocessing. In bucket i, all items have weight at most $\overline{w}_i := \max_{j \in B_i} w_j \le 2^{-i}$, and, with the exception of bucket L, at least $\overline{w}_i/2$. PEs generate geometrically distributed skip values $v := \lfloor \ln(\texttt{rand}())/\ln(1-\overline{w}_i) \rfloor$ and then consider the $j := v/\overline{w}_i$-th item. The algorithm then uses rejection to output the item with probability $w_j/\overline{w}_i$. This process is repeated until a PE exceeds its allotted item range. In all buckets $i < L$, the acceptance probability is at least $1/2$ for every element, leading to an efficient algorithm in expectation. In bucket L, the smaller acceptance probability is not a problem, as with high probability only a constant number of items is ever considered to begin with. This is because items in bucket L have probability at most $1/n$. In total, this requires work $\mathcal{O}(1+W)$ in expectation.

Although all PEs have to do about the same expected amount of work, there are some random fluctuations between the actual amount of work depending on the actual exponential variates that are computed. Using Chernoff bounds once more, these can be bounded to $\mathcal{O}(W/p + \log n)$ with high probability – leading to a (conservative) $\log n$ term for the span.

Each PE returns an array containing its part of the sample. ◀

9.1 Distributed Case

For the distributed setting, observe that Problem WRS–S can be solved entirely independently over disjoint subsets of the input if we are not interested in load balancing. No communication is needed. We thus can directly adapt the result from Bringmann et al. [6]. Note that for a PE with n_i items, it suffices to sort them into $\log n_i$ categories now which is possible in linear time. A query on a PE with total weight W_i will take expected time $\mathcal{O}(1+W_i)$. As expected parallel query time we get $\mathcal{O}(\max_i W_i + \log n)$ using an argument analogous to the proof of Theorem 15.

Once more, we analyze load balancing for the case that the items are distributed randomly.

▶ **Theorem 16.** *When items are distributed randomly, subset sampling (Problem WRS–S) can be done in a communication-free way with expected preprocessing overhead $\mathcal{O}(n/p + \log p)$ and expected sampling time $\mathcal{O}(W/p + \log p)$.*

Proof. The bound for the preprocessing follows by applying Chernoff bounds to the distribution of the number of items. The query bound follows in a similar way by exploiting that the expected maximum load is maximized when all the weight is concentrated in W items of weight 1 [28]. ◀

10 Sampling with a Reservoir

We adapt the streaming algorithm of Efraimidis et al. [12] to a distributed mini-batch streaming model (also known as *discretized streams*), where PEs process variable-size batches of items one at a time. The PEs' memory is too small to store previous batches, only the current mini-batch is available in memory. This is a generalization of the traditional data stream model and widely used in practice, *e.g.*, in Apache Spark Streaming [36], where it is called *discretized streams*.

First, we show how to adapt sampling with skip values (exponential jumps) of Efraimidis et al. [12, Section 4] to the exponential variates described in Section 2.3. This allows for

faster and numerically more efficient generation in practice. The difference between the variates associated with items in Efraimidis et al. [12] and here is a simple $x \mapsto -\ln(x)$ mapping. Because of the sign inversion, the reservoir R contains the elements with the *smallest* associated variates. Let v_i denote the key of item i, that is, the exponentially distributed variate associated with it, and define $T := \max_{i \in R} v_i$ as the threshold value, *i.e.,* the largest key of any item in the reservoir. Then the skip value X describing the total amount of weight to be skipped can be computed as $X := -\ln(\texttt{rand}())/T$. The key associated with the item j that is to be inserted into the reservoir is then $v_j := -\ln(\texttt{rand}(e^{-Tw_j}, 1))/w_j$, where $\texttt{rand}(a, b) := a + \texttt{rand}()(b - a)$ generates a uniform random variate from the interval $[a, b)$. The range of this variate has been chosen so that v_j is less than T (as at this step in the algorithm, it has already been determined that item j must be part of the reservoir, we need to compute a suitable variate from the distribution associated with its weight). It then replaces the item with the largest key in the reservoir, and the threshold T is updated accordingly.

We now show how to use this sequential algorithm to construct a distributed weighted reservoir sampler.

▶ **Theorem 17.** *For WRS–B with sample size k, consider mini-batches consisting of up to b elements per PE. Processing such a mini-batch is possible in $\mathcal{O}\big(b + b^* \log(b^* + k) + \alpha \log^2(kp)\big)$ time, where $b^* \leq b$ is the maximum number of items from the mini-batch on any PE that is below the insertion threshold.*[7]

If the items' weights are positive and independently drawn from a common continuous distribution, every PE processes n/p elements, and all batches have b elements per PE, then this algorithm inserts $\mathcal{O}(\frac{k}{p}(1 + \log \frac{n}{k}))$ items into each local reservoir in expectation.

Proof. We maintain the local reservoirs using augmented search trees that support splitting in logarithmic time [22]. When processing a mini-batch of size b, we proceed as in the sequential algorithm above and insert new elements into the reservoir. However, the insertion threshold T remains unchanged (initially, and as long as less than k elements are in the global reservoir, $T = -\infty$). By definition of b^*, these insertions require time $\mathcal{O}(b^* \log(b^* + k))$ in total. Since we have to process each element's weight even when using the above skip value technique, $\mathcal{O}(b)$ time is required to identify the items to be inserted into the reservoir. Once all PEs have finished processing their batch, we compute the new threshold. Using a distributed selection operation on the search trees, this can be implemented in expected time $\mathcal{O}\big(\alpha \log^2(kp)\big)$. We then discard the items with keys exceeding the new threshold in $\mathcal{O}(\log(k + b^*))$ time using a *split* operation on the local search tree.

The question now is how many items we unnecessarily insert into the local reservoirs. Efraimidis et al. [12, Proposition 7] show that if the w_i are independent random variates from a common continuous distribution, their sequential reservoir sampling algorithm inserts $\mathcal{O}(k \log(n/k))$ items into the reservoir in expectation. We adapt this to mini-batches of b elements per PE. Let X_i denote the number of insertions on a PE for batch i. We obtain

$$\mathbf{E}[X_i] = \sum_{j=1}^{b} \mathbf{P}[\text{item } j \text{ is inserted}] = b \cdot \frac{k}{n_{\text{pre}}} \leq b \cdot \frac{k}{ipb} = \frac{k}{ip},$$

[7] The latency term can be reduced to $\mathcal{O}(\log(kp))$ if we are willing to let the sample size vary in the range $k..2k$ using the approximate sorted selection algorithm from [15]. If kp is very large, we can also use the unsorted selection algorithm of [15] with latency $\mathcal{O}(\log n)$ but higher communication volume.

where n_{pre} is the number of elements seen globally before the batch began. For the initial $i_0 = \frac{k}{bp}$ iterations this probability exceeds one and we account for this with b insertions per PE – overall $b \cdot \frac{k}{bp} = k/p$. For minibatches $i_0 \leq i < \frac{n}{pb}$ we obtain

$$\mathbf{E}\left[\sum X_i\right] \leq \sum_{\frac{k}{bp} \leq i \leq \frac{n}{bp}} \frac{k}{ip} = \frac{k}{p} \sum_{\frac{k}{bp} \leq i \leq \frac{n}{bp}} \frac{1}{i} = \frac{k}{p}(H_{\frac{n}{bp}} - H_{\frac{k}{bp}}) \leq \frac{k}{p}(1 + \ln \frac{n}{bp} - \ln \frac{k}{bp}) = \frac{k}{p}(1 + \ln \frac{n}{k})$$

where H_n is the n-th harmonic number. ◀

11 Experiments

We performed strong and weak scaling experiments on 2-alias tables from Section 4 and on a shared-memory implementation of "distributed" 2-level alias tables from Theorem 3-(1) (Section 5).

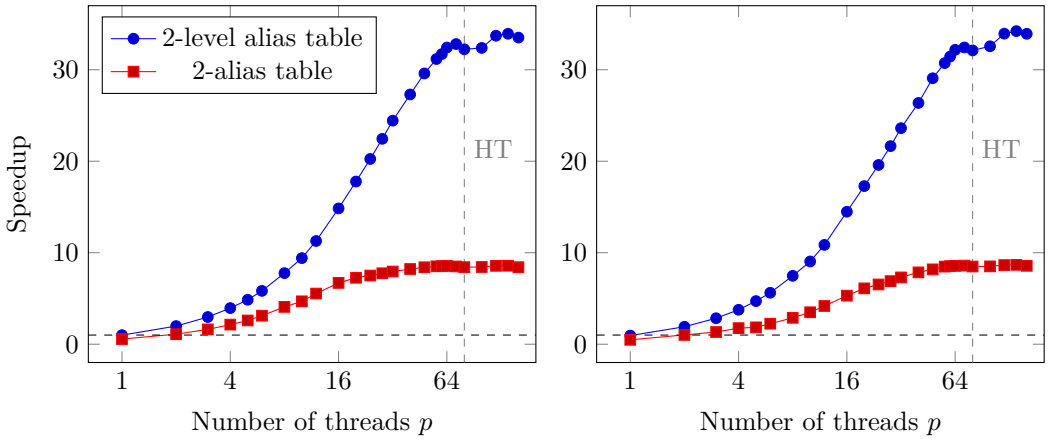
Hardware Environment

All experiments were conducted on a machine with four Intel Xeon Gold 6138 CPUs (4×20 cores, 160 hyper-threads) and 768 GiB of DDR4-2666 main memory, running Ubuntu 18.04. The CPUs are fully connected with Intel UPI links. Each CPU has six memory channels, for a total of 24 memory channels. The high number of cores necessitates some degree of Non-Uniform Memory Access (NUMA) awareness in memory-bound applications like ours. Thus, all large arrays are distributed over the available NUMA nodes, and threads are assigned to maintain data locality.

Implementation Details

We made a C++ implementation of the 2-alias tables from Section 4 and of the 2-level alias tables from Section 5 (using shared memory). Our implementation is publicly available under the GNU General Public License (version 3) at https://github.com/lorenzhs/wrs. We also implemented sequential 1-alias tables as a baseline. A surprising difference between our sequential implementation compared to existing ones (*e.g.,* the GNU Scientific Library's (GSL) `gsl_ran_discrete_preproc` function or the R project for statistical computing's `sample` function) is memory layout of the alias table data structure. Instead of using the struct-of-arrays paradigm, we use an array of structs. This greatly increases memory locality, incurring one instead of up to two cache misses per query. As a result, our sequential implementation's query time is 2.18× faster than that of GSL version 2.4 ($n = 10^7$, average over 10 iterations of 20 repetitions with 10^7 queries each), while also achieving 1.17× faster construction time. Other popular statistics packages, such as NumPy (version 1.5.1, function `np.choice`) or Octave (Statistics package version 1.4.0, function `randsample`) employ algorithms with $\omega(k)$ query running time. We therefore use our own implementation of 1-alias tables as the baseline in our evaluation.

Compared to the descriptions in Sections 2.2 and 4, we performed some minor additional modifications to construction of the tables to further improve performance. In the 1-alias table, we store triplets $b = (w_i, i, a_i)$ of a weight w_i, item i and alias a_i. While occupying additional space, this allows for an optimization at query time, where we return $b[1 + \texttt{rand}() >= w_i]$, saving a conditional branch. In 2-alias tables, buckets consist of two cumulative probabilities and up to three items. The first item to be placed in a bucket, whether a light item or a piece of a heavy item, is always placed in the first slot, and the associated probability in the



(a) Strong scaling with input size $n = 10^9$. **(b)** Weak scaling, per-thread input size $n/p = 10^7$.

Figure 4 Strong and weak scaling evaluation of parallel alias table construction techniques. Speedups are measured relative to an optimized sequential implementation of 1-alias tables.

first probability slot. When a second item is placed in the bucket, the second probability slot is assigned the sum of the two item's probabilities, as this addition would otherwise have to occur at query time.

Construction

Figure 4 shows the results of our scaling experiments. In Figure 4a, the strong scaling speedup achieved compared to an optimized sequential implementation of 1-alias tables on $n = 10^9$ uniformly random input values is shown. The gains of parallelization are limited by the available memory bandwidth. Because the construction of 2-alias tables requires multiple passes over the data, speedups do not increase further once the machine's memory bandwidth is saturated. In contrast, 2-level alias tables can be constructed almost independently by the PEs and require much fewer accesses to memory. Hyper-threading (HT) yields only negligible additional speedup for both approaches.

The results of a weak scaling experiment using per-thread input size $n/p = 10^7$ is shown in Figure 4b. Despite the input size being much smaller over most of the number of threads, we obtain nearly identical speedup values. Again, memory bandwidth is the limiting factor.

Queries

Queries of our 2-alias tables incur an average slowdown of 22 % compared to our optimized implementation of 1-alias tables. The shared-memory implementation of our distributed approach suffers a slowdown of 44 % since an additional query for a meta-item is needed. Since the sequential implementation used as a baseline is 2.18× faster than the GNU Scientific Library's 1-alias table implementation, the query times of both of our parallel implementations are still significantly faster than what is widely used.

12 Conclusions and Future Work

We have presented parallel algorithms for a wide spectrum of weighted sampling problems running on variety of machine models. The algorithms are at the same time efficient in

theory and sufficiently simple to be practically useful.

Future work can address implementations of further algorithms and support of additional machine models such as MapReduce or of other big data tools like Thrill [3] or Spark [37]. For example, the solution to WRS–1 based on Theorem 3-(3) could be implemented on top of Thrill using its prefix sum primitive. It might also be possible to transfer some of the distributed data structures. For example, the variant of Theorem 3-(1) could be supported by emulating the behavior of $p = \sqrt{n}$ PEs. Storing the $\sqrt{n}$ second-level tables as elementary objects in the big data tool ensures load balancing and fault tolerance; a replicated meta-table of size $\sqrt{n}$ can be used to assign samples to groups.

We can probably improve the work bound of WRS–B by exploiting integer keys for the distributed priority queue. We could also study additional variants of reservoir sampling, *e.g.,* in a sliding window setting or sampling with replacement. An interesting open problem is whether 1-alias tables (with a single alias per bucket) can be computed efficiently in parallel.

References

1 Joachim H. Ahrens and Ulrich Dieter. Sequential random sampling. *ACM Transactions on Mathematical Software (TOMS)*, 11(2):157–169, June 1985.

2 Richard Arratia. On the amount of dependence in the prime factorization of a uniform random integer. *Contemporary combinatorics*, 10:29–91, 2002. Page 36.

3 Timo Bingmann, Michael Axtmann, Emanual Jöbstl, Sebastian Lamm, Huyen Chau Nguyen, Alexander Noe, Sebastian Schlag, Matthias Stumpp, Tobias Sturm, and Peter Sanders. Thrill: High-performance algorithmic distributed batch data processing with C++. In *2016 IEEE International Conference on Big Data*, pages 172–183. IEEE, 2016.

4 Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38(11):1526–1538, Nov 1989.

5 Karl Bringmann and Kasper Green Larsen. Succinct sampling from discrete distributions. In *45th ACM Symposium on Theory of Computing (STOC)*, pages 775–782. ACM, 2013.

6 Karl Bringmann and Konstantinos Panagiotou. Efficient sampling methods for discrete distributions. *Algorithmica*, 79(2):484–508, 2017.

7 M. T. Chao. A general purpose unequal probability sampling plan. *Biometrika*, 69(3): 653–656, 1982.

8 Richard Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

9 Luc Devroye. *Non-Uniform Random Variate Generation*. Springer, 1986.

10 Pavlos S Efraimidis. Weighted random sampling over data streams. In *Algorithms, Probability, Networks, and Games: Scientific Papers and Essays Dedicated to Paul G. Spirakis on the Occasion of His 60th Birthday*, pages 183–195. Springer, 2015.

11 Pavlos S Efraimidis and Paul G Spirakis. Fast parallel weighted random sampling. Technical Report TR99.04.02, CTI Patras, 1999.

12 Pavlos S Efraimidis and Paul G Spirakis. Weighted random sampling with a reservoir. *Information Processing Letters*, 97(5):181–185, 2006.

13 Torben Hagerup. Fast parallel generation of random permutations. In *18th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 405–416. Springer, 1991.

14 Torben Hagerup, Kurt Mehlhorn, and J Ian Munro. Maintaining discrete probability distributions optimally. In *20th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 253–264. Springer, 1993.

15 Lorenz Hübschle-Schneider and Peter Sanders. Communication efficient algorithms for top-k selection problems. In *30th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 659–668. IEEE, 2016.

16 Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison Wesley, 1992.

17 Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing. Design and Analysis of Algorithms*. Benjamin/Cummings, 1994.

18 Kevin J Lang. Practical algorithms for generating a random ordering of the elements of a weighted set. *Theory of Computing Systems*, 54(4):659–688, 2014.

19 George Marsaglia, Wai Wan Tsang, Jingbo Wang, et al. Fast generation of discrete random variables. *Journal of Statistical Software*, 11(3):1–11, 2004.

20 Yossi Matias, Jeffrey Scott Vitter, and Wen-Chun Ni. Dynamic generation of discrete random variates. *Theory of Computing Systems*, 36(4):329–358, 2003.

21 Jens Maue and Peter Sanders. Engineering algorithms for approximate weighted matching. In *6th Workshop on Experimental Algorithms (WEA)*, pages 242–255. Springer, 2007.

22 Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures — The Basic Toolbox*. Springer, 2008.

23 Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

24 Kirill Müller. Accelerating weighted random sampling without replacement. *Arbeitsberichte Verkehrs-und Raumplanung*, 1141, 2016.

25 Frank Olken and Doron Rotem. Random sampling from databases: a survey. *Statistics and Computing*, 5(1):25–42, 1995.

26 Sanguthevar Rajasekaran and John H Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM Journal on Computing*, 18(3):594–607, 1989.

27 Abhiram G Ranade. How to emulate shared memory. *Journal of Computer and System Sciences*, 42(3):307–326, 1991.

28 Peter Sanders. On the competitive analysis of randomized static load balancing. In S. Rajasekaran, editor, *First Workshop on Randomized Parallel Algorithms*, Honolulu, Hawaii, 1996. http://algo2.iti.kit.edu/sanders/papers/rand96.pdf.

29 Peter Sanders. Random permutations on distributed, external and hierarchical memory. *Information Processing Letters*, 67(6):305–310, 1998.

30 Peter Sanders, Sebastian Schlag, and Ingo Müller. Communication efficient algorithms for fundamental big data problems. In *2013 IEEE International Conference on Big Data*, pages 15–23. IEEE, 2013.

31 Peter Sanders, Sebastian Lamm, Lorenz Hübschle-Schneider, Emanuel Schrade, and Carsten Dachsbacher. Efficient random sampling – parallel, vectorized, cache-efficient, and online. *ACM Transaction on Mathematical Sofware*, 44(3):29:1–29:14, 2018.

32 Julian Shun. Improved parallel construction of wavelet trees and rank/select structures. In *2017 Data Compression Conference (DCC)*, pages 92–101. IEEE, 2017.

33 Michael D. Vose. A linear algorithm for generating random numbers with a given distribution. *IEEE Transactions on Software Engineering (TSE)*, 17(9):972–975, 1991.

34 Alastair J Walker. An efficient method for generating discrete random variables with general distributions. *ACM Transactions on Mathematical Software (TOMS)*, 3(3): 253–256, 1977.

35 Chak-Kuen Wong and Malcolm C. Easton. An efficient method for weighted sampling without replacement. *SIAM Journal on Computing*, 9(1):111–113, 1980.

36 Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 423–438. ACM, 2013.

37 Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache Spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.