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, permutations, subsets, and reservoirs. We also give improved sequential algorithms for alias table construction and for sampling with replacement. Experiments on shared-memory parallel machines with up to 158 threads show near linear speedups both for construction and queries.

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 categorical distribution, multinoulli distribution, parallel algorithm, alias method, PRAM, communication efficient algorithm, subset sampling, reservoir sampling

Digital Object Identifier 10.4230/LIPIcs.ESA.2019.59

Related Version A full version of the paper is available at `https://arxiv.org/abs/1903.00227`.

Supplement Material The code and scripts used for our experiments are available under the GPLv3 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. [26], Olken et al. [28]). 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 $\{i, \ldots, j\}$. Item i has weight w_i and $W := \sum_{i=1}^{n} w_i$. Define $u := \log U$ where $U := w_{\max}/w_{\min} := \max_i w_i / \min_i w_i$.

27th Annual European Symposium on Algorithms (ESA 2019).
Editors: Michael A. Bender, Ola Svensson, and Grzegorz Herman; Article No. 59; pp. 59:1–59:24
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

WRS–1: Weighted sampling of *one item* from a categorical (or multinoulli) distribution (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.

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 variants with 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.

Outline

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 consider Problem WRS–1. We first give an improved sequential algorithm for constructing

Table 1 Result overview (expected and asymptotic). Distributed results assume random distribution of inputs. 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. The complexity of sorting n integers with keys from $0..x$ is $\text{isort}_x(n)$ ($\text{isort}^* =$ parallel, $\text{isort}^1 =$ sequential).

	Shared Memory					Distributed		
		Preprocessing		Query			Preprocessing	Query
Problem	§	Work	Span	Work	Span	§	Time	Time
WRS–1	4.2	n	$\log n$	1	1	4.3	$\frac{n}{p} + \alpha \log p$	α
WRS–R	5	$\text{isort}^*_u(n)$		$s + \log n$	$\log n$	5.1	$\text{isort}^1_u(\frac{n}{p}) + \alpha \log p$	$\frac{s}{p} + \log p$
WRS–N	6	$\text{isort}^*_u(n)$		$k + \log n$	$\log n$	6	$\text{isort}^1_u(\frac{n}{p}) + \alpha \log p$	$\frac{k}{p} + \alpha \log^2 n$
WRS–N						6	$\frac{n}{p} + \beta u + \alpha \log p$	$\frac{k}{p} + \alpha \log n$
WRP	7	—	—	$\text{isort}^*_{n(u+\log n)}(n)$		7	—	$\text{isort}^*_{n(u+\log n)}(n)$
WRS–S	8	n	$\log n$	$s + \log n$	$\log n$	8	$\frac{n}{p} + \log p$	$\frac{s}{p} + \log p$
WRS–B	—	—	—	—	—	9	—	$b \log(b + k) + \alpha \log^2 kp$

alias tables – the most widely used data structure for Problem WRS–1 that allows sampling in constant time. Then we parallelize that algorithm for shared and distributed memory. We also present parallel construction for a more space efficient variant.

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 5 develops such an algorithm which is interesting both as a parallel and a sequential algorithm.

Section 6 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 7 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 [20], 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 [24].

For subset sampling (Problem WRS–S), we parallelize the approach of Bringmann et al. [6] in Section 8. 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 9, 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 [16].

Section 10 gives a detailed experimental evaluation of our algorithms for WRS–1 and WRS–R. Section 11 summarizes the results and discusses possible future directions.

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 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 GPGPU.

For shared-memory algorithms we use the CREW PRAM model (concurrent read exclusive write parallel random access machine) [18]. 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 $\mathcal{O}(\beta p + \alpha \log p)$ [19]. We will particularly strive to obtain *communication-efficient algorithms* [35] 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 [30, Lemma 3.1]. For a CRCW PRAM, expected linear work and logarithmic span can be achieved when $K = \mathcal{O}(n \log^c n)$ [30] (the paper gives the constraint $K = \mathcal{O}(n)$ but the generalization is obvious and important for us in Section 7). 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/m. 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. [28]).

Deterministic sampling with only a single memory probe is possible using Walker's alias table method [38], and its improved construction due to Vose [37]. An alias table consists of $m := n$ buckets where bucket $b[i]$ represents some part w'_i of the weight of item i. The remaining weight of the heavier items is distributed to the remaining capacity of the buckets such that each bucket only represents one other item (the *alias* a_i). Algorithm 1 gives high-level pseudocode for the approach proposed by Vose. The items are first classified into light and heavy items. Then the heavy items are distributed over light items until their residual weight drops below W/n. They are then treated in the same way as light items.

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 $b[r].a$ for tails.

Algorithm 1 Classical construction of alias tables similar to Vose's approach [37].

```
Procedure voseAliasTable(⟨w_1, ..., w_n⟩, b : Array of w : ℝ × a : ℕ)
    W := Σ_i w_i                                          −− total weight
    h := {i ∈ 1..n : w_i > W/n}  : Stack                  −− heavy items
    ℓ := {i ∈ 1..n : w_i ≤ W/n}  : Stack                  −− light items
    for i := 1 to n do  b[i].w := w_i                     −− init buckets with weights
    while h ≠ ∅ do                                        −− consume heavy items
        j := h.pop                                        −− get a heavy item
        while b[j].w > W/n do                             −− still heavy
            i := ℓ.pop                                    −− get a light item
            b[i].a := j                                   −− Fill bucket b[i] with a ...
            b[j].w := (b[j].w + b[i].w) − W/n             −− ... piece of item j.
        ℓ.push(j)                                         −− Bucket j is light now.
```

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 [11, 12, 10]. Equivalently, one can generate exponential random variates $v_i := -\ln(\texttt{rand}())/w_i$ and 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 [34]. 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 [38, 37, 21, 9, 6]. 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 [15, 22].

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 $\mathcal{O}\big(k \log \frac{n}{k}\big)$ [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 [40, 28, 15, 22]. With the most efficient such algorithms [15, 22] 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 [20] 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 [27].

3.3 Parallel Sampling

There is surprisingly little work on parallel sampling. Even uniform unweighted sampling had many loose ends until recently [34]. Parallel uniformly random permutations are covered in [14, 33]. 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 solving the selection problem suffices if the output need not be sorted. The optimal dynamic data structure for WRS–1 [22] admits a parallel bulk update in the (somewhat exotic) combining-CRCW-PRAM model. However, this does not help with Problem WRS–N since batch sizes are one.

4 Alias Table Construction (Problem WRS–1)

4.1 Improved Sequential Alias Tables

Before discussing parallel alias table construction, we discuss a simpler, faster and more space efficient sequential algorithm that is a better basis for parallelization. Previous algorithms need auxiliary arrays/queues of size $\Theta(n)$ in order to decide in which order the buckets are filled. Vose [37] mentions that this can be avoided but does not give a concrete algorithm. We now describe an algorithm with this property.

The idea of the algorithm is that two indices i and j sweep the input array with respect to light and heavy items, respectively. The loop invariant is that the weight of items corresponding to light (heavy) items preceding i (j) has already been distributed over some buckets and that their corresponding buckets have already been constructed. Variable w stores the weight of the part of item j that has not yet been assigned to buckets. Each iteration of the main loop advances one of the indices and initializes one bucket. When the residual weight w exceeds W/n, item j is used to fill bucket i, the residual weight w is reduced by $W/n - w_i$, and index i is advanced to the next light item. Otherwise, the remaining weight of heavy item j fits into bucket j and the remaining capacity of bucket j is filled with the next heavy item. Algorithm 2 gives pseudocode that emphasizes the high degree of symmetry between these two cases.

Algorithm 2 A sweeping algorithm for building alias tables.

```
Procedure sweepingAliasTable(⟨w_1, ..., w_n⟩, b : Array of w : ℝ × a : ℕ)
    W := Σ_i w_i                                        −− total weight
    i := min {k > 0 : w_k ≤ W/n}                        −− first light item
    j := min {k > 0 : w_k > W/n}                        −− first heavy item
    w := w_j                                            −− residual weight of current heavy item
    while j ≤ n do
        if w > W/n then                                 −− Pack a light bucket.
            b[i].w := w_i                               −− Item i completely fits here.
            b[i].a := j                                 −− Item j fills the remainder of bucket i.
            w := (w + w_i) − W/n                        −− Update residual weight of item j.
            i := min {k > i : w_k ≤ W/n}                −− next light item, assume w_{n+2} = 0
        else                                            −− Pack a heavy bucket.
            b[j].w := w                                 −− Now item j completely fits here.
            b[j].a := j' := min {k > j : w_k > W/n}     −− next heavy item, assume w_{n+1} = ∞
            w := (w + w_{j'}) − W/n                     −− Find residual weight of item j'.
            j := j'
```

4.2 Parallel Alias Tables

The basic idea behind our *splitting based algorithm* is to identify subsets L and H of light ($w_i \leq W/n$) and heavy ($w_i > W/n$) items such that they can be allocated precisely within their respective buckets, *i.e.,* $w(H \cup L) := \sum_{i \in H \cup L} w_i = (|H| + |L|) \cdot W/n$. By splitting the items into such pairs of subsets, we can perform alias table construction for these subsets in parallel. Since the above balance condition cannot always be achieved, we allow to "steal" a piece of a further heavy item, *i.e.,* this item can be used to fill buckets in several subproblems. Such a split item will only be used as an alias except in the last subproblem where it is used. Thus, the computed data structure is still an alias table.

We first explain how to split n items into two subsets of size n' and $n - n'$. Similar to Vose's algorithm, we first compute arrays ℓ and h containing the indices of the light and heavy items, respectively. We then determine indices i and j such that $i + j = n'$, $\sigma := \sum_{x \leq i} w_{\ell[x]} + \sum_{x \leq j} w_{h[x]} \leq n'W/n$ and $\sigma + w_{h[j+1]} > n'W/n$; see Figure 1 for an example. These values can be determined by binary search over the value of j. By precomputing prefix sums of the weights of the items in ℓ and h, each iteration of the binary search takes constant time. The resulting subproblem then consists of the light items $L := \{\ell[1], \ldots, \ell[i]\}$, the heavy items $H := \{h[1], \ldots, h[j]\}$ and a fraction of size $n'W/n - \sigma$ of item $h[j+1]$.

To split the input into p independent subproblems of near-equal size, we perform the above two-way-split for the values $n'_k = \lceil nk/p \rceil$ for $k \in 1..p-1$. PE k is then responsible for filling a set of buckets corresponding to sets of light and heavy items, each represented by a range of indices into ℓ and h. A piece of a further heavy item may be used to make the calculation work out. Note that a subproblem might contain an empty set of light or heavy items and that a single heavy item j may be assigned partially to multiple subproblems, but only the last PE using a heavy item will fill its bucket.

Algorithm 3 gives detailed pseudocode. It uses function split to compute $p - 1$ different splits in parallel. The result triple (i, j, s) of split specifies that buckets $\ell[1] \cdots \ell[i]$ as well as $h[1] \cdots h[j]$ shall be filled using the left subproblem. Moreover, total weight s of item $h[j+1]$ is *not used* on the left side, *i.e.,* spilled over to the right side.

This splitting information is then used to make p parallel calls to procedure pack – giving each PE the task to fill up to $\lceil n/p \rceil$ buckets. Pack has input parameters specifying ranges of heavy and light items it should use. The parameter spill determines how much weight of item $h[\underline{j} - 1]$ can be used for that. Pack works similar to the sweeping algorithm from Algorithm 2. If the residual weight of item $h[\underline{j} - 1]$ drops below W/n, this item is actually also packed in this call. The body of the main loop is dominated by one if-then-else case distinction. When the residual weight of the current heavy item falls below W/n, its bucket is filled using the next heavy item. Otherwise, its weight is used to fill the current light item.

▶ **Theorem 1.** *We can construct an alias table with work $\Theta(n)$ and span $\Theta(\log n)$ on a CREW PRAM.*

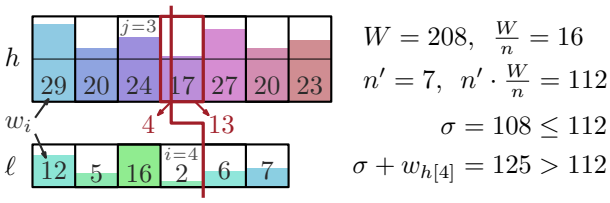


Figure 1 Parallel alias tables: splitting $n = 13$ items into two parts of size $n' = 7$ and $n - n' = 6$.

Algorithm 3 Pseudocode for parallel splitting based alias table construction (PSA).

```
Procedure psaAliasTable(⟨w_1, ..., w_n⟩, b : Array of w : ℝ × a : ℕ)
    W := Σ_i w_i                                                    -- total weight
    h := {i ∈ 1..n : w_i > W/n}  : Array        -- parallel traversal finds heavy items and
    ℓ := {i ∈ 1..n : w_i ≤ W/n}  : Array                            -- and light items
    for k := 1 to p − 1 dopar (i_k, j_k, spill_k) := split(⌈nk/p⌉)   -- split into p pieces
    (i_0, j_0, spill_0) := (0, 0, 0);   (i_p, j_p) := (n, n)        -- cover corner cases
    for k := 1 to p dopar pack(i_{k−1} + 1, i_k,   j_{k−1} + 1, j_k,   spill_{k−1})

    Function split(n') : ℕ × ℕ × ℝ
        a := 1;   b := min(n', |h|)                                 -- a..b is search range for j
        loop                                                        -- binary search
            j := ⌊(a + b)/2⌋                                        -- bisect search range
            i := n' − j                                  -- Establish the invariant i + j = n'.
            σ := Σ_{x≤i} w_{ℓ[x]} + Σ_{x≤j} w_{h[x]}  -- work to the left; use precomputed prefix sums
            if σ ≤ n'W/n and σ + w_{h[j+1]} > n'W/n then return (i, j, w_{h[j+1]} + σ − n'W/n)
            if σ ≤ n'W/n then b := j − 1   else a := j + 1          -- narrow search range

    (* pack buckets b[ℓ[i̲]], ..., b[ℓ[ī]] and buckets b[h[j̲]], ..., b[h[j̄]]. *)
    (* Use up to weight spill from item h[j̲ − 1]. *)
    Procedure pack(i̲, ī,   j̲, j̄,   spill)
        i := i̲                                                      -- ℓ[i] is the current light item.
        j := j̲ − 1                                                  -- h[j] is the current heavy item.
        w := spill                                       -- part of current heavy item still to be placed
        if spill = 0 then j++;   w := w_{h[j]}
        loop
            if w ≤ W/n then                                         -- pack a heavy bucket
                if j > j̄ then return
                b[h[j]].w := w
                b[h[j]].a := h[j + 1]
                w := (w + w_{h[j+1]}) − W/n
                j++
            else                                                    -- pack a light bucket
                if i > ī then return
                b[ℓ[i]].w := w_{ℓ[i]}
                b[ℓ[i]].a := h[j]
                w := (w + w_{ℓ[i]}) − W/n
                i++
```

Proof. The algorithm requires linear work and logarithmic span for identifying light and heavy items and for computing prefix sums [4] over them. Splitting works in logarithmic time. Then each PE needs time $\mathcal{O}(n/p)$ to fill the buckets assigned to its subproblem. ◀

4.3 Distributed Alias Table Construction

The parallel algorithm described in Section 4.2 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, 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.[1] 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 2.** *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*

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

$$\alpha \log^2 p \text{ expected time using the algorithm from Section 4.2} \tag{2}$$

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

Proof. Building the local 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.[2] Equation (2) then follows by using the shared-memory algorithm from Section 4.2. It can be implemented to run in expected time $\mathcal{O}\big(\alpha \log^2 p\big)$ on a distributed-memory machine using PRAM emulation [31].

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 4.4. 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 $\mathcal{O}(\alpha \log p)$. ◀

4.3.1 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 alias tables. This implies a trade-off between redistribution cost (part of preprocessing) and load balance during sampling.

We now look at the case where an adversary can choose an arbitrarily skewed distribution of item sizes but where the items are randomly allocated to PEs (or that we actively randomize the allocation implying $\mathcal{O}(n/p)$ additional communication volume).

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

[2] If we ensure that meta-items have similar size (see Section 4.3.1) then we can arrange the meta-items in such a way that $i = j$ most of the time.

▶ **Theorem 3.** *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 time $\mathcal{O}\big(\alpha \log^2 p\big)$ when supporting deterministic queries (Theorem 2-(2)) and expected time $\mathcal{O}(\alpha \log p)$ using rejection sampling (Theorem 2-(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 $\mathcal{O}(W/p)$ [32].

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 2-(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 $\mathcal{O}\big(\log^3 p\big)$ 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 $\mathcal{O}\big(\alpha \log^2 p\big)$ 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 2-(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. ◀

4.4 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.

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 [36] 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 4.** *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 Output Sensitive Sampling With Replacement (Problem WRS–R)

The algorithm of Section 4.2 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.

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 [34], 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 weights differ by at most a factor of two – weight w_i is mapped to group $\lfloor \log(w_i/w_{\min}) \rfloor$.

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)* 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.[3]

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:

[3] If desired, we can also avoid rejection sampling by mapping the items without gaps into up to $2n_G$ buckets of size a. This way there are at most two items in each bucket. Note that this is still different from alias tables because we need to map ranges of consecutive items to ranges of buckets. This is not possible for alias tables.

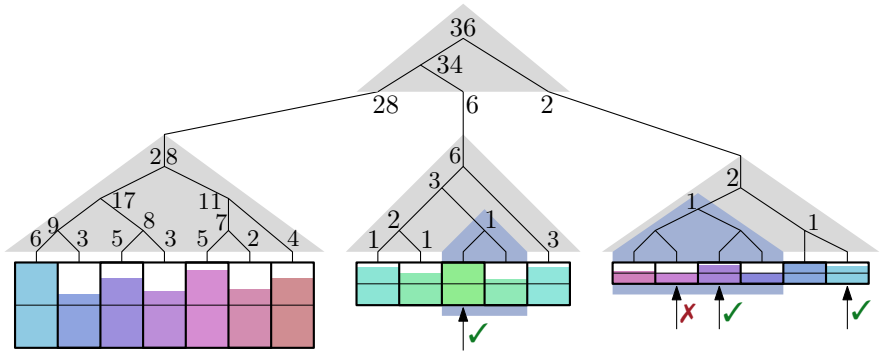


Figure 2 Output-sensitive sampling: assignment of multiplicities with $k = 36$.

▶ **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*[4] *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 4.2.

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

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 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. ◀

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

▶ **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

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

Hence,

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

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

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

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

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

5.1 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 [34]. 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.,* [19]). 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 4.3, 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* $\mathcal{O}\big(\mathrm{isort}_u^1(n/p) + \alpha \log p\big)$ *where* $\mathrm{isort}_u^1(x)$ *denotes the time for sequential integer sorting of* x *elements using keys from the range* $0..u$.[5] *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, 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. ◀

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

We can construct an algorithm for sampling without replacement based on the output-sensitive algorithm for sampling *with* replacement of Section 5. Presume we know an $\ell > k$ so that a sample of size ℓ with replacement contains at least k and no more than $\mathcal{O}(k)$ unique items. Then we can obtain a sample with $k' \geq k$ different items using the algorithm of Section 5, discard the multiplicities, and downsample to size k using the exponential clocks method (see Section 2.3).

To find the right value for ℓ, we derive an estimation of the number of unique samples as a function of ℓ. The basis of this estimation is to assume that sufficiently heavy items are sampled once and lighter items are sampled with probability proportional to their weight. We precompute the data needed for the estimation for each group and then perform a binary search over the groups. More concretely, when the currently considered group stores elements with weights in the range $[a, 2a)$, we try the value $\ell = \lceil 1/(2a) \rceil$. We (over)estimate the resulting number of unique samples as

$$|\{i : w_i \geq a\}| + \ell \cdot \frac{\sum \{w_i : w_i < a\}}{W} \quad .$$

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

In the full paper we show that this is a good estimate and how to use it to drive the binary search.

7 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 a lot is known about parallel sorting. The downside 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.,* [25, 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.

In the full paper, we explain how to achieve the same for the highly skewed distribution needed for WRP by applying radix sort and the monotonous transformation function $f(r, w_i) := n \ln(-\ln(r) n w_{\max}/w_i)$.

8 Subset Sampling (Problem WRS–S)

Subset sampling is a generalization of Bernoulli Sampling to the weighted case. The unweighted case can be solved in expected time linear in the output size by computing the geometrically distributed distances between elements in the sample [1]. The naïve algorithm for the weighted problem, 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.

The difference between WRS–S on the one hand and WRS–1/WRS–R on the other hand is that we do not have a fixed sample size but rather treat the item weights as independent inclusion probabilities in the sample (this requires $w_i \leq 1$ for all i). Hence, 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 $\mathcal{O}(1 + W)$.

In the full paper we parallelize the approach of Bringmann et al. [6]. Similar to our algorithm for sorting with replacement, this algorithm is based on grouping items into sets with similar weight. In each group, one can use ordinary Bernoulli sampling in connection with rejection sampling. Load balanced division between PEs can be done with a prefix sum calculation over the weights in each group.

9 Sampling with a Reservoir

In the full paper, we adapt the streaming algorithm of Efraimidis et al. [12] to a distributed mini-batch streaming model, 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 [41], where it is called *discretized streams*. The basic idea is to keep the reservoir in a distributed priority queue [16].

10 Experiments

We now report experiments on alias tables (Problem WRS–1, Section 4) and the closely related problem of sampling with replacement (Problem WRS–R, Section 5).

Experimental Setup. We use machines with Intel and AMD processors in our experiments. The Intel machine has four Xeon Gold 6138 CPUs (4×20 cores, 160 hyper-threads, of which we use up to 158 to minimize the influence of system tasks on our measurements) and 768 GiB of DDR4-2666 main memory. The AMD machine is a single-socket system with a 32-core AMD EPYC 7551P CPU (64 hyper-threads, of which we use up to 62) and 256 GiB of DDR4-2666 RAM. While single-socket, this machine also has non-uniform memory access (NUMA), as the CPU consists of four dies internally. Both machines run Ubuntu 18.04. All implementations are in C++ and compiled with GNU `g++` 8.2.0 (flags `-O3 -flto`).

Our measurements do not include time spent on memory allocation and mapping.

Implementation Details. We implemented alias table construction using our parallel splitting algorithm (PSA, Section 4.2) and output-sensitive sampling with replacement (OS, Section 5), as well as a shared-memory version of the distributed algorithm (2lvl, Section 4.3, Theorem 2-(1)). The 2lvl algorithm can either use Vose's method (2lvl-classic, Algorithm 1) or our sweeping algorithm of Section 4.1 (2lvl-sweep, Algorithm 2) as base case. For OS, we use an additional optimization that aborts the tree descent and uses the base case bucket table when fewer than 128 samples are to be drawn from at least half as many items. The resulting elements are then deduplicated using a hash table to ensure that each element occurs only once in the output. A variant without this deduplication is called OS-ND and may be interesting if items *may* be returned multiple times. We also implemented sequential versions of both alias table construction methods. Both of the machines used require some degree of Non-Uniform Memory Access (NUMA) awareness in memory-bound applications like ours. Thus, in our parallel implementations, all large arrays are distributed over the available NUMA nodes, and threads are pinned to NUMA nodes to maintain data locality. All pseudorandom numbers are generated with 64-bit Mersenne Twisters [23], using the Intel Math Kernel Library (MKL) [17] on the Intel machine and `dSFMT`[6] on the AMD machine. All of our implementations are publicly available under the GNU General Public License (version 3) at `https://github.com/lorenzhs/wrs`.

Compared to the descriptions in Sections 2.2 and 4.2, we performed a minor modification to construction of the tables to improve query performance. In the alias table, we store tuples $(w_i, A = [i, a_i])$ of a weight w_i, item i and alias a_i. This allows for an optimization at query time, where we return $A[\texttt{rand}() \cdot W/n \geq w_i]$, saving a conditional branch. When indices are 32-bit integers and weights are 64-bit doubles, this does not use additional space since the record size is padded to 128 bits anyway.

Sequential Performance. Surprisingly, many common existing implementations of alias tables (*e.g.,* `gsl_ran_discrete_preproc` in the GNU Scientific Library (GSL) [13] or `sample` in the R project for statistical computing [29]) use a struct-of-arrays memory layout for the alias table data structure. By using the array-of-structs paradigm instead, we can greatly improve memory locality, incurring one instead of up to two cache misses per query. Combined with branchless selection inside the buckets and a faster random number generator,

[6] `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/`, version 2.2.3

our query is more than three times as fast as that of GSL version 2.5 (measured for $n = 10^8$). At the same time, alias table construction using our implementation of Vose's method is 30 % faster than GSL. 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 superlinear query time. We therefore use our own implementation of Vose's algorithm as the baseline in our evaluation.

Among our sequential implementations, construction with Vose's method is slightly faster than our sweeping algorithm. On the Intel machine, it is 8 % faster, while on the AMD machine, the gap is 3 %. However, since all of our measurements exclude the time for memory allocations, this is not the full picture. If we include memory allocation, our method is around 5 % faster than Vose's on both machines. This is because it requires no additional space, compared to $\mathcal{O}(n)$ auxiliary space for Vose's method.

The optimization described above to make queries branchless lowers query time substantially, namely by 22 % on the Intel machine and 27 % on the AMD machine, again for $n = 10^8$. Storing the item indices at construction time comes at no measurable extra cost.

10.1 Construction

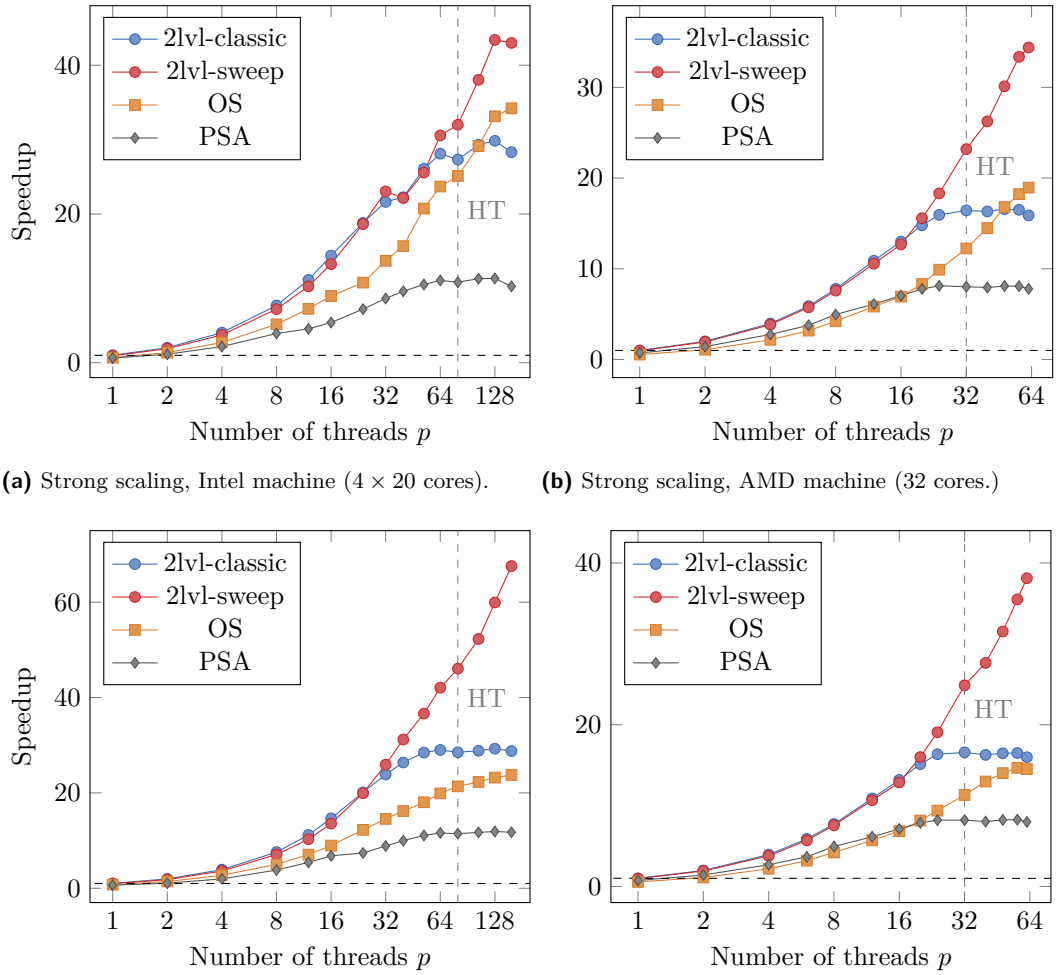
Speedups compared to an optimized sequential implementation of Vose's alias table construction algorithm are shown in Figure 3 (strong scaling with $n = 10^8$ and weak scaling with $n/p = 10^7$ uniform random variates). Speedups do not increase further once the machine's memory bandwidth is saturated, limiting the speedup that can be achieved with techniques that require multiple passes over the data (PSA, 2lvl-classic). In contrast, 2lvl-sweep can be constructed almost independently by the PEs and requires much fewer accesses to memory. Sequentially, there is little difference between our sweeping algorithm and Vose's method. However, our algorithm scales much better to high thread counts because it reduces the memory traffic and since hyper-threading (HT) helps to hide the overhead of branch mispredictions. This is especially visible on the AMD machine (Figure 3b), where 2lvl-sweep achieves more than twice the speedup of 2lvl-classic, 34 compared to 16. The lack of scaling for 2lvl-classic and 2lvl-sweep when going from 32 to 40 cores on the Intel machine (Figure 3a) coincides with a large frequency reduction in the CPUs at this point [39]. Preprocessing for OS introduces some overhead but is not much slower than 2lvl.

In the weak scaling experiments (Figures 3c and 3d), we again see clearly how 2lvl-classic and PSA are limited by memory bandwidth. Using more than two threads per available memory channel (4×6 for the Intel machine, 8 for the AMD machine) yields nearly no additional benefit for these algorithms. Meanwhile, 2lvl-sweep and OS are not limited by the available memory bandwidth, but rather latency of memory accesses. As a result, they scale well even to the highest thread counts.

10.2 Queries

We performed strong and weak scaling experiments for queries (Figure 4) as well as throughput measurements for different sample sizes (Figure 5). Besides uniform random variates, we use random permutations of the weights $\{1^{-s}, 2^{-s}, \cdots, n^{-s}\}$ for a parameter s to obtain more skewed "power-law" distributions.

Scaling. First, consider the scaling experiments of Figure 4. These experiments were conducted on the Intel machine, as its highly non-uniform memory access characteristics highlight the differences between the algorithms. All speedups are given relative to sampling



(a) Strong scaling, Intel machine (4×20 cores).

(b) Strong scaling, AMD machine (32 cores.)

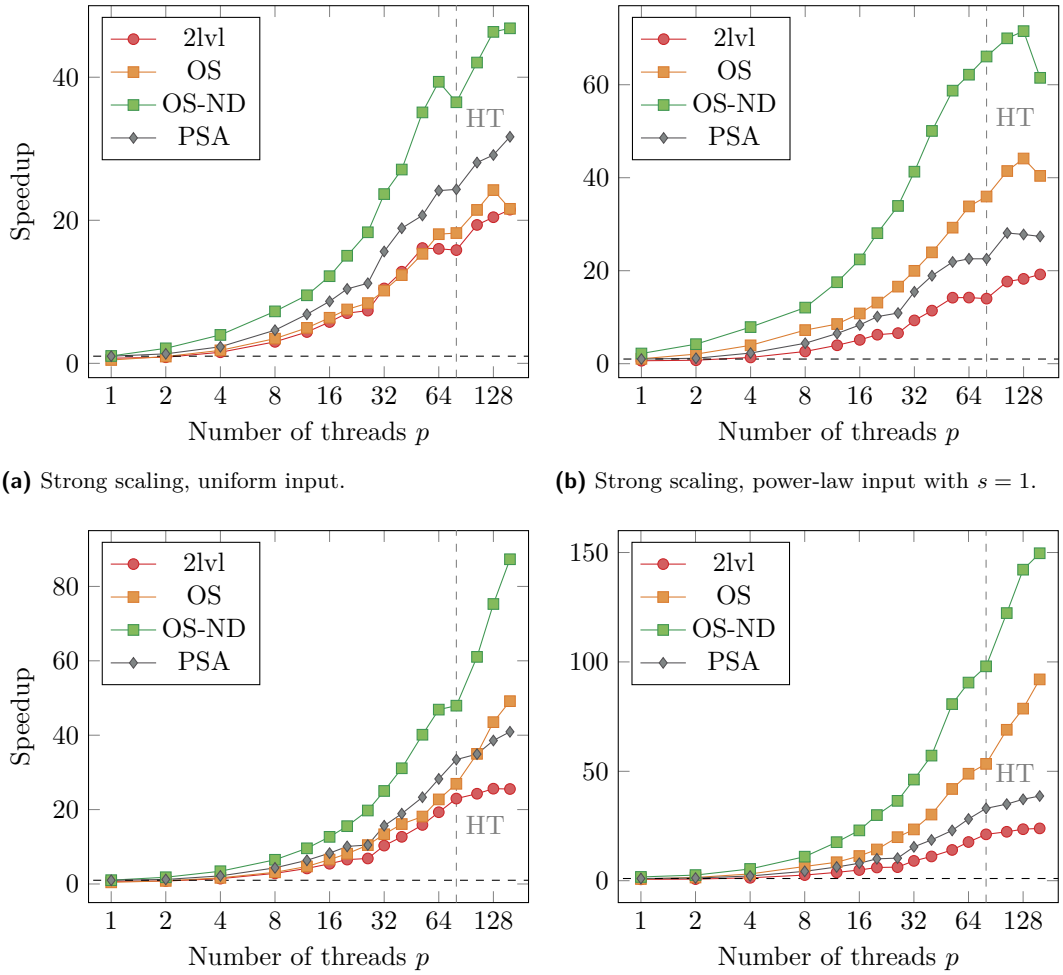
(c) Weak scaling, Intel machine (4×20 cores).

(d) Weak scaling, AMD machine (32 cores).

Figure 3 Strong (top) and weak (bottom) scaling evaluation of parallel alias table construction techniques. Strong scaling with input size $n = 10^8$, weak scaling with $n/p = 10^7$. Speedups are measured relative to our optimized implementation of Vose's method (Algorithm 1, Section 2.2).

sequentially from an alias table. The strong scaling experiments (Figures 4a and 4b) deliberately use a small sample size to show scaling to low per-thread sample counts ($\approx 64\,000$ for 158 threads). We can see that all algorithms have good scaling behavior. Hyper-threading (marked "HT" in the plots) yields additional speedups, as it helps to hide memory latency. This already shows that sampling is bound by random access latency to memory. Sampling from PSA and 2lvl is done completely independently by all threads, with no interaction apart from reading from the same shared data structures. Because the Intel machine has four NUMA zones, most queries have to access another NUMA node's memory. This limits the speedups achievable using PSA and 2lvl.

On the other hand, OS and OS-ND have a shared top-level sample assignment stage, after which threads only access local memory. This benefits scaling, especially on NUMA machines. As a result, OS-ND achieves the best speedups, despite this benchmark producing very few samples with multiplicity greater than one (Figure 4a, sample size is 1 % of input size). On the other hand, deduplication in the base case of OS has significant overhead, making it roughly 25 % slower than sampling from an alias table for such inputs, even sequentially.



(a) Strong scaling, uniform input.

(b) Strong scaling, power-law input with $s = 1$.

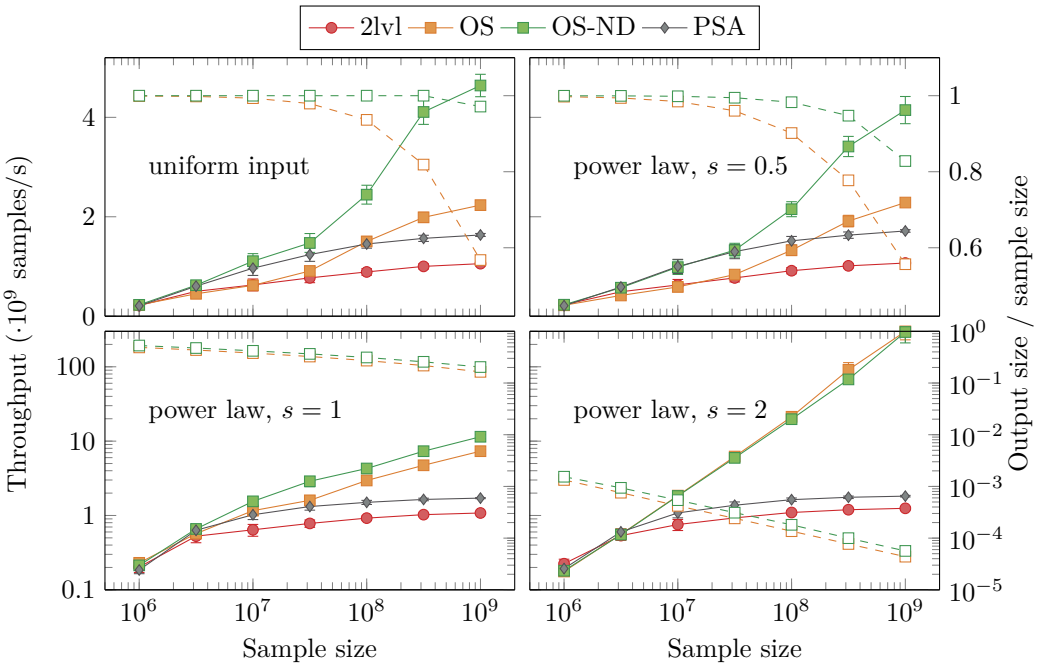
(c) Weak scaling, uniform input.

(d) Weak scaling, power-law input with $s = 1$.

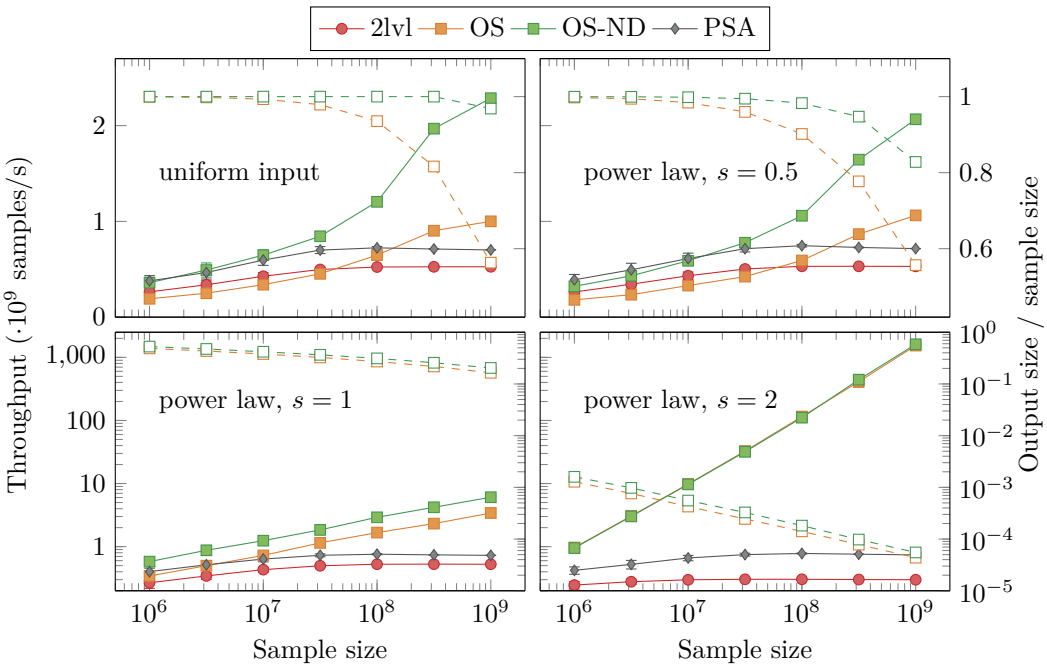
Figure 4 Query strong and weak scaling for $n = 10^9$ input elements. Sample size for strong scaling $s = 10^7$, per-thread sample size for weak scaling $s/p = 10^6$. All speedups relative to sequential alias tables. Intel machine.

The weak scaling experiments of Figures 4c and 4d show even better speedups because many more samples are drawn here than in our strong scaling experiment, reducing overheads. Sampling from a classical alias table (PSA) achieves a speedup of 40 here, again limited by memory accesses rather than computation. Meanwhile, the output-sensitive methods (OS, OS-ND) reap the benefits of accessing only local memory.

Throughput. Figure 5 shows the query throughput of the different approaches. We can see that 2lvl suffers a significant slowdown compared to PSA on all inputs since an additional query for a meta-item is needed (this is also clearly visible in Figure 4). This slowdown is much more pronounced on the Intel machine (60 %) than on the AMD machine (30 %) because inter-NUMA-node memory access latency on the quad-socket Intel machine is much higher than on the single-socket AMD machine. Nonetheless, throughput is limited by the latency of random accesses to memory for the (bottom) tables for both approaches and on both machines.

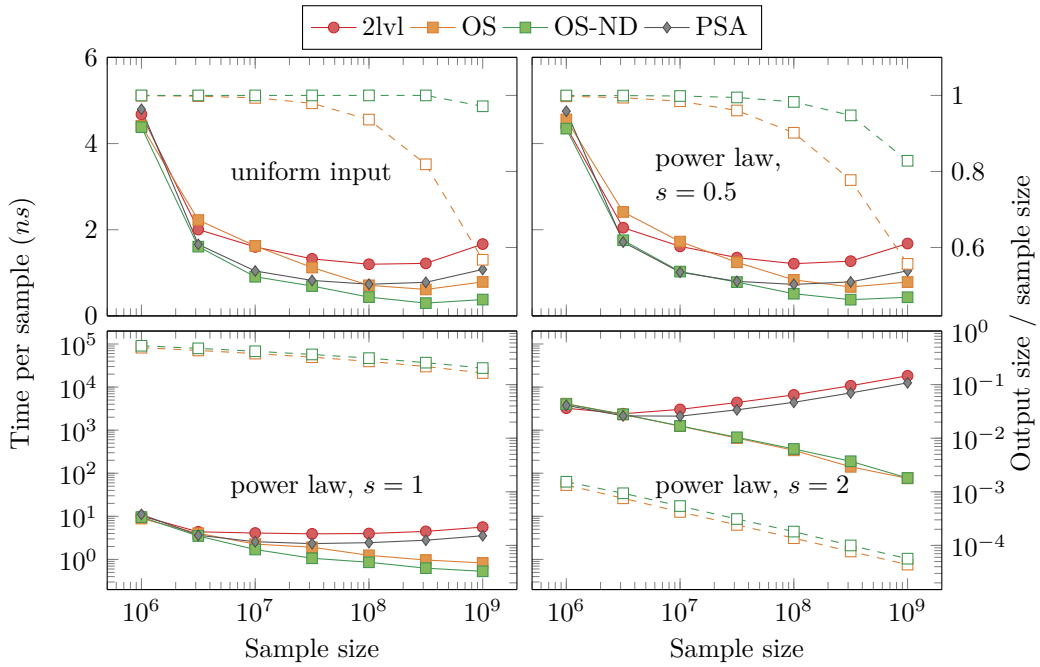


(a) Intel machine (158 threads). Note the logarithmic y axes for the bottom plots.

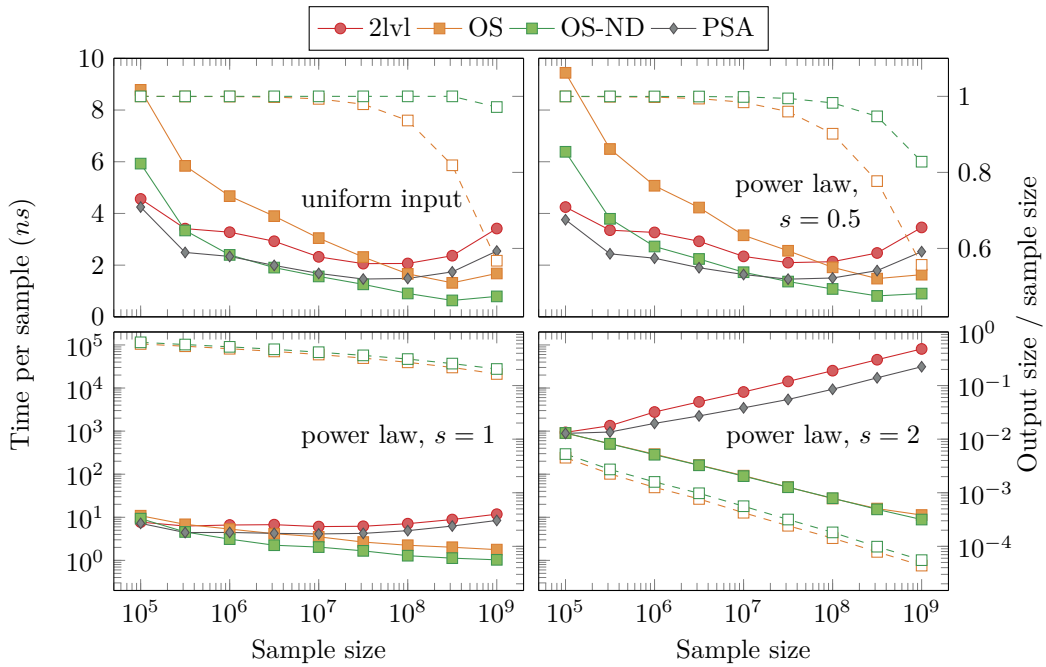


(b) AMD machine (62 threads). Note the logarithmic y axes for the bottom plots.

Figure 5 Query throughput of the different methods for $n = 10^9$, using all available cores. Top left: uniform inputs, top right: power law with $s = 0.5$, bottom left: power law $s = 1$, bottom right: power law $s = 2$. Dashed lines on the right y axis belong to the same-colored solid lines on the left y axis and show fraction of output size over sample size for output-sensitive algorithms.



(a) Intel machine (158 threads). Note the logarithmic y axes for the bottom plots.



(b) AMD machine (62 threads). Note the logarithmic y axes for the bottom plots.

Figure 6 Time per unique output item for the different methods for $n = 10^9$ using all available cores on the Intel (top) and AMD (bottom) machines. Top left: uniform inputs, top right: power law with $s = 0.5$, bottom left: power law $s = 1$, bottom right: power law $s = 2$. Dashed lines on the right y axis belong to the same-colored solid lines on the left y axis and show fraction of output size over sample size for output-sensitive algorithms.

As long as the sample contains few duplicates (*cf.* the dashed lines with scale on the right y axis, which belong to the solid lines of the same color and marker shape), the cost of base case deduplication in OS cancels out the gains of increased memory locality. On the AMD machine, where memory access locality is less important, this results in higher throughput for 2lvl than for OS for small sample sizes when inputs are not too skewed. As expected, when there are many duplicates, the output sensitive algorithms (OS and OS-ND) scale very well. Omitting base case deduplication (OS-ND) doubles throughput for uniform inputs and does no harm for skewed inputs, making OS-ND the consistently fastest algorithm. In comparison, adding sequential deduplication to normal alias tables using a fast hash table (Google's `dense_hash_map`) takes 5.4 times longer for uniform inputs ($n = 10^8$, 10^7 samples) compared to simply storing samples in an array without deduplication.

Lastly, we observe that 2lvl and PSA throughput levels off after $10^{7.5}$ samples on the AMD machine, whereas it keeps increasing slightly on the Intel machine.

Time per Item. Figure 6 shows the time per unique item in the sample. We can see that the 2lvl and PSA approaches work well as long as few items have multiplicity larger than one. In these cases, what OS gains from having higher locality of memory accesses is lost in base case deduplication – especially on the AMD machine. Because it may emit items repeatedly, OS-ND does not suffer from this and is the fastest algorithm. The same is true for the power law inputs with $s = 0.5$ and $s = 1$ (observe that as in Figure 5, the y axes for the lower two plots are logarithmic). For power law inputs with $s = 2$, the running time of OS and OS-ND is nearly constant regardless of sample size . This is because the number of unique items is very low for this input (measured in the low thousands), and thus what little time is spent on sampling is dominated by thread synchronization and scheduling overhead. This overhead is particularly problematic with the 158 threads on the Intel machine (Figure 6a), where it amounts to several milliseconds, ten times as much as on the AMD machine (Figure 6b). Further, observe that the leveling off of 2lvl and PSA throughput on the AMD machine causes unfavorable time per item for large samples.

11 Conclusions and Future Work

We have presented parallel algorithms for a wide spectrum of weighted sampling problems running on a 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 the trade-off between parallel alias tables and (distributed) 2-level alias tables (fast queries versus fast scalable construction). We can also consider support of additional machine models such as GPUs as well as MapReduce or of other big data tools like Thrill [3] or Spark [42]. For example, the solution to WRS–1 based on Theorem 2-(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 2-(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.

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, November 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 Mark Galassi, Jim Davies, James Theiler, Brian Gough, Gerard Jungmann, Patrick Alken, Michael Booth, Fabrice Rossi, and Rhys Ulerich. *GNU scientific library: reference manual*. Network Theory, 3 edition, 2009.

14 Torben Hagerup. Fast Parallel Generation of Random Permutations. In *18th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 405–416. Springer, 1991.

15 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.

16 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.

17 Intel. *Intel Math Kernel Library 2019*. Intel, 2019. URL: `https://software.intel.com/en-us/mkl-reference-manual-for-c`.

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

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

20 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.

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

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

23 M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACMTMCS: ACM Transactions on Modeling and Computer Simulation*, 8:3–30, 1998.

24 Jens Maue and Peter Sanders. Engineering Algorithms for Approximate Weighted Matching. In *6th Workshop on Experimental Algorithms (WEA)*, pages 242–255. Springer, 2007.

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

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

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

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

29 R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2019. URL: `https://www.R-project.org`.

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

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

32 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`.

33 Peter Sanders. Random Permutations on Distributed, External and Hierarchical Memory. *Information Processing Letters*, 67(6):305–310, 1998.

34 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.

35 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.

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

37 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.

38 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.

39 Wikichip.org. Intel Xeon Gold 6138. `https://en.wikichip.org/w/index.php?title=intel/xeon_gold/6138&oldid=71062`, 2019. Accessed April 26, 2019.

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

41 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.

42 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.