Brief Announcement: Communication-Efficient Weighted Reservoir Sampling from Fully Distributed Data Streams

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

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

ABSTRACT

We consider weighted random sampling from distributed data streams presented as a sequence of mini-batches of items. This is a natural model for distributed streaming computation, and our goal is to showcase its usefulness. We present and analyze a fully distributed, communication-efficient algorithm for weighted reservoir sampling in this model. An experimental evaluation on up to 256 nodes (5120 processors) shows good speedups, while theoretical analysis promises further scaling to much larger machines.

CCS CONCEPTS

• **Theory of computation → Sketching and sampling**; **Parallel algorithms**; *Data structures design and analysis.*

KEYWORDS

sampling, weighted sampling, reservoir sampling, mini-batch, data stream, communication efficiency

ACM Reference Format:
Lorenz Hübschle-Schneider and Peter Sanders. 2020. Brief Announcement: Communication-Efficient Weighted Reservoir Sampling from Fully Distributed Data Streams. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '20), July 15–17, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3350755.3400287

1 INTRODUCTION

We consider the problem of maintaining a random sample without replacement over an input set that is revealed over time in small batches and in a distributed fashion. The items have weights associated with them and are part of the sample with probability proportional to their share of the total weight. Our design goal is to minimize communication between the nodes (a detailed motivation of communication efficiency is given in [12, 19]), while our overall goal is to showcase the usefulness of the mini-batch model.

Problem Definition. Let the input consist of n items, which we shall refer to by their indices from $1..n$[1], the weight of item i be $w_i \in \mathbb{R}_+$, and let $W := \sum_{i=1}^{n} w_i$ denote the total weight. The items are processed in batches of variable size. After processing a batch, update the weighted random sample without replacement of size $\min(k, n')$ of all n' items seen up to and including the current batch.

A *weighted random sample without replacement* of size k consists of k different items $s_1 \neq \ldots \neq s_k$ so that for $j \in 1..k$ and $i \notin \{s_\ell \mid \ell < j\}$, $\mathbf{P}\left[s_j = i\right] = w_i / \left(W - \sum_{\ell<j} w_{s_\ell}\right)$. Note that a second definition of the problem exists where the probability of each item to be *included in* the sample is proportional to its relative weight. There, items with weight $w_i/W > 1/k$ have inclusion probability greater than one (*infeasible* items) and require special treatment, see *e.g.*, [9].

1.1 Related Work

For an overview of the broader literature on random sampling, refer to [17] for the uniform case and [13] for weighted sampling. Here, we limit ourselves to the literature on reservoir sampling.

Uniform sampling from data streams has been studied since at least the early 1960s [11]. A simple folklore method is to associate uniform random deviates with the items, retaining the k items with the smallest associated values. Several asymptotically optimal algorithms are known [16, 22]. Their key insight is that it is possible to compute the distance between two samples in constant time [8, 16]. More recently, sampling from the union of multiple data streams has also received some attention [4, 6, 7, 21]. However, in addition to assuming synchronous operation of the nodes and the network, the *distributed streaming* (or *continuous distributed monitoring*) *model* used therein relies on a centralized coordinator node which stores the entire sample and is the exclusive communication partner of all other nodes, severely limiting scalability in practice. An algorithm in a shared-memory mini-batch model was presented recently [20].

For *weighted* items, Chao [3] presents an elegant algorithm for an alternative definition of weighted sampling (see above). Mapping weights to exponential (instead of uniform) deviates allows for a simple selection of the k smallest values to obtain a weighted sample [1, 9, 10]. A reduction to sampling with replacement eliminates the effects of numerical inaccuracies of floating-point representations [2]. To our knowledge, only [15] considers weighted distributed reservoir sampling. It uses the distributed streaming model, resulting in challenges orthogonal to those we face here.

2 PRELIMINARIES

Machine Model. We have p processing elements (PEs) numbered $1..p$, connected by a network with full-duplex, single-ported communication. Sending a message of length ℓ takes time $\alpha + \beta\ell$, where α is the time to initiate the transfer and β the subsequent transmission time per machine word. Treating α and β as variables in asymptotic analysis allows us to combine *internal work* x, *communication volume* y and *latency* z into a single term: $O(x + \beta y + \alpha z)$.

[1] $a..b$ is shorthand for $\{a, \ldots, b\}$ throughout this paper.

SPAA '20, July 15–17, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-6935-0/20/07.
https://doi.org/10.1145/3350755.3400287

Selection from Sorted Sequences. Our algorithm relies on selection from the union of sorted sequences stored locally at the PEs. Which selection algorithm to use depends on the data and requirements. Let n bound per-PE input size and k be the rank of the desired item. In the general case, we can use an algorithm with expected running time $O(\log(kp) \cdot \log n + \alpha \log^2(kp))$ [12, Section IV-B]. If k is allowed to vary in some range $\underline{k}..\overline{k}$ with $\overline{k} - \underline{k} \in \Omega(\overline{k}/d)$ for some $d \in \mathbb{N}$, it is possible to find an item with rank between $\underline{k}$ and $\overline{k}$ in expected time $O(d \log n + \beta d + \alpha \log p)$ [12, Lemma 3]. Further options are detailed in our technical report [14].

3 MINI-BATCH MODEL

Our model is a batched view on streaming algorithms. Items arrive as a series of *mini-batches* on small time intervals. For example, each mini-batch could be defined as the set of all items that arrived within a certain time window (*e.g.,* in Apache Spark Streaming [23]). Because memory is limited, only the current mini-batch is available in memory at each point in time. This is a generalization of other models of streaming algorithms: on a sequential underlying machine with batch size 1, we obtain the sequential streaming model (see, *e.g.,* [20]). In a distributed model with p sites (nodes) which exchange fixed-size messages with a coordinator, batch size 1 yields the distributed streaming model, also known as the *continuous distributed monitoring* model [5]. In this paper, we use the distributed message-passing model described in Section 2.

Unless explicitly specified, we shall make no assumptions about the distribution of mini-batch sizes across PEs or over time, nor about the distribution of items. In algorithm analysis, we denote by b the maximum number of items in the current batch at any PE, and by B the sum of all PEs' current batch sizes. Thus, an algorithm expressed in the mini-batch model can handle arbitrarily imbalanced inputs without any impact on correctness; however, load balance may suffer if the number of items per PE differs widely.

4 WEIGHTED RESERVOIR SAMPLING

The basic idea of our algorithm is to combine the exponential clocks method—associating with each item an exponentially distributed key using the item's weight for the rate parameter [1, 9, 10, 13]—with a communication-efficient bulk priority queue to maintain the set of the k items with the smallest keys, *i.e.,* the sample. Each PE is solely responsible for the items that were seen in its input, and no PE gets a special role (such as a coordinator node). During each batch, a PE inserts into its local reservoir all items whose key is smaller than the largest key of any item in the sample (the global *threshold*). When a batch finishes, the PEs perform a distributed selection for the k-th smallest key, which becomes the new global threshold, and discard all items with larger keys. The remaining elements form the new sample. During a mini-batch, the threshold remains unchanged.

Our algorithm adapts the sequential skip distance calculation of Efraimidis and Spirakis [10, Section 4] with an $x \mapsto -\ln(x)$ mapping, which improves numerical accuracy and simplifies generation. Details are shown in Algorithm 1 and our technical report [14].

The reservoir is maintained in a distributed fashion, with each PE's *local reservoir* represented as a B+ tree augmented with *split*, *rank*, and *select* support (see, *e.g.,* [18, Sections 7.3.2 and 7.5.2]).

Algorithm 1 Pseudocode for Weighted Reservoir Sampling

```
Input: A the local part of the mini-batch of items, T the previous batch's
    threshold (initially −∞), R the local reservoir (initially empty)
Output: The new threshold and the updated local reservoir
def processBatch(A : Item[], T : ℝ, R : Reservoir) : ℝ × Reservoir
    Item: ℝ+ × ℕ with weight w ∈ ℝ+, index i ∈ ℕ
    Reservoir: B+ tree mapping keys from ℝ to item IDs
    if T < 0 then                          — fewer than k items seen globally before
        foreach (w, i) ∈ A do              — exponentially distributed keys
            R.insert(−ln(rand())/w, i)
    else
        j := 0 : ℕ;                        — 1-based index of next item, initially invalid
        while j ≤ |A| do
            X := −ln(rand())/T : ℝ         — weight to be skipped
            while X > 0 do                 — skip X amount of weight in total
                j := j + 1;
                if j > |A| then break from both loops
                X := X − A[j].w
            y := exp(−T · A[j].w) : ℝ
            v := −ln(y + rand()(1 − y))/A[j].w : ℝ      — new key
            R.insert(v, A[j].i)
    (T, i) := select(R, k)                 — select k globally smallest and new threshold
    (R, _) := R.splitAt(i)                 — discard local items with larger keys
    return (T, R)                          — return new threshold and reservoir
```

At the end of a mini-batch, the PEs jointly select the globally k-th smallest key (see Section 2) in the union of all local reservoirs, which becomes the insertion threshold for the next batch. Each PE then discards all items with larger keys using a *split* operation. The remaining k items in the union of all local reservoirs form the desired sample of all items seen so far. Algorithm 1 gives pseudocode.

Theorem 4.1. *A mini-batch of up to b items per PE can be processed in time $O(b + (b^* + 1)\log(b^* + k) + T_{sel})$, where $b^* \le b$ is the maximum number of items from the mini-batch below the insertion threshold on any PE, and T_{sel} is the time for selection from sorted sequences of size at most $b^* + k$ per PE (see Section 2).*

Proof. By definition of b^*, the local insertions require time $O(b^* \log(b^* + k))$ in total because each local reservoirs has size at most k at the start of the batch. Since we have to process each item's weight even when using skip distances, $O(b)$ time is required to identify the items to be inserted into the reservoir. The selection operation takes time T_{sel} which varies depending on the specifics of the input. The number of candidate items per PE for the selection is clearly bounded by the local reservoir size of at most $k + b^*$. The split operation to discard the items with keys exceeding the new threshold takes time logarithmic therein, *i.e.,* $O(\log(k + b^*))$. □

We now consider how many items we (unnecessarily) insert into local reservoirs by keeping the threshold fixed during a mini-batch. The next theorem, which we prove in the technical report [14], can be viewed as an average case of Theorem 4.1.

Theorem 4.2. *If all items' weights are independently drawn from the same continuous distribution and all batches have the same number of items on every PE, then our algorithm inserts no more than $O\left(\frac{k}{p}\log\frac{n}{k} + \log p\right)$ items into* any *local reservoir in expectation.*

5 EXPERIMENTS

We implemented our algorithm with the multi-pivot approximate selection algorithm of [12, Section IV-C], using 8 pivots and exact bounds ($\underline{k} = \overline{k} = k$).[2] We compare it to a centralized algorithm which uses the same thresholding procedure but gathers all candidate items at a designated root PE, where it uses sequential selection to maintain the sample, subsequently labeled *gather*. The code, available at https://github.com/lorenzhs/reservoir, is in C++ and was compiled with g++ 8.2.0 and OpenMPI 4.0. It was executed on up to 256 nodes of ForHLR II, an HPC cluster with two 10-core Intel Xeon E5-2660 v3 CPUs per node. Each core is one PE (20 PEs per node). We use uniformly random weights from the interval $(0, 100]$ as input. Each PE receives the same number of items per mini-batch.

The results of a weak scaling experiment with per-PE batch size $b = 10^6$ (left) and 10^5 (right) items and sample sizes of $k = 10^3$, 10^4, and 10^5 are given in Fig. 1, showing the speedup relative to our algorithm on a single node ($p = 20$). We can see that our algorithm shows good scaling across the board, and, as expected, smaller samples are slightly faster to maintain than larger ones. Clearly, the centralized algorithm performs well only for small samples, struggling even with $k = 10^4$ for the larger batch size and performing badly regardless of batch size for $k = 10^5$. Both algorithms achieve better—and for our algorithm, near-optimal—speedups for large batch sizes, as communication overhead is more noticeable for small batches, where local processing is fast.

A running time composition analysis confirms that for $k = 10^5$, local processing dominates our algorithms' running time for larger batch sizes, whereas the centralized algorithm spends most of its time on selection and, as p grows, gathering the candidate items.

We also conducted strong scaling experiments (not shown here due to space limitations), which confirm consistent scaling of our method as long as per-PE batch sizes do not drop below around 10^4. More details on our strong and weak scaling experiments as well as a running-time composition analysis are given in our technical report [14].

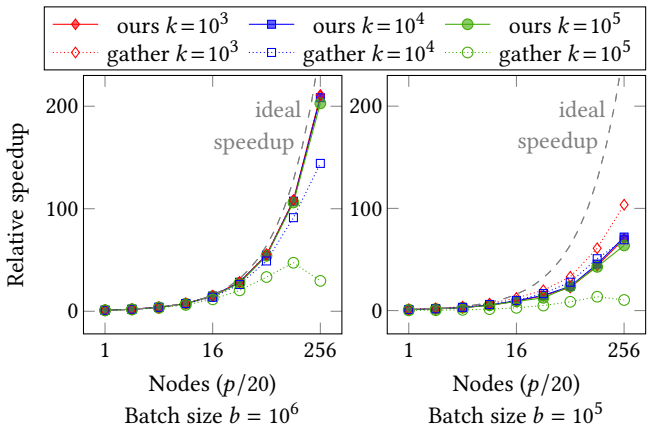

Figure 1: Weak scaling, speedup relative to a single node (p=20) with our algorithm

[2] In this configuration, its asymptotical running time matches the non-approximate algorithm [12, Section IV-B], but its pivot selection speeds up convergence in practice.

6 CONCLUSION

We presented an efficient weighted reservoir sampling algorithm as a showcase of the usefulness of the distributed mini-batch model of streaming algorithms. Analysis and experiments show that our algorithm performs well in theory as well as in practice.

Note that our algorithm can easily be modified to handle unweighted (uniform) inputs by using the well-known skip distances for uniform reservoir sampling [8, p. 640], which saves the $O(b)$ term in Theorem 4.1 because X items can be skipped in $O(1)$ time.

Acknowledgment. This work was performed on the supercomputer ForHLR funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research.

REFERENCES

[1] Richard Arratia. 2002. On the amount of dependence in the prime factorization of a uniform random integer. *Contemporary combinatorics* 10 (2002), 29–91. p.36.
[2] Vladimir Braverman, Rafail Ostrovsky, and Gregory Vorsanger. 2015. Weighted sampling without replacement from data streams. *Inform. Process. Lett.* 115, 12 (2015), 923–926.
[3] M. T. Chao. 1982. A general purpose unequal probability sampling plan. *Biometrika* 69, 3 (1982), 653–656.
[4] Yung-Yu Chung, Srikanta Tirthapura, and David P. Woodruff. 2016. A Simple Message-Optimal Algorithm for Random Sampling from a Distributed Stream. *IEEE Trans. Knowl. Data Eng.* 28, 6 (2016), 1356–1368.
[5] Graham Cormode. 2013. The continuous distributed monitoring model. *ACM SIGMOD Record* 42, 1 (2013), 5–14.
[6] Graham Cormode, S Muthukrishnan, Ke Yi, and Qin Zhang. 2010. Optimal sampling from distributed streams. In *29th ACM Symposium on Principles of Database Systems (PODS '10)*. 77–86.
[7] Graham Cormode, S Muthukrishnan, Ke Yi, and Qin Zhang. 2012. Continuous sampling from distributed streams. *J. ACM* 59, 2 (2012), 10.
[8] Luc Devroye. 1986. *Non-Uniform Random Variate Generation*. Springer.
[9] Pavlos S. Efraimidis. 2015. Weighted random sampling over data streams. In *Algorithms, Probability, Networks, and Games*. LNCS, Vol. 9295. Springer, 183–195.
[10] Pavlos S. Efraimidis and Paul G. Spirakis. 2006. Weighted random sampling with a reservoir. *Inform. Process. Lett.* 97, 5 (2006), 181–185.
[11] CT Fan, Mervin E Muller, and Ivan Rezucha. 1962. Development of sampling plans by using sequential (item by item) selection techniques and digital computers. *J. Amer. Statist. Assoc.* 57, 298 (1962), 387–402.
[12] Lorenz Hübschle-Schneider and Peter Sanders. 2016. Communication Efficient Algorithms for Top-k Selection Problems. In *30th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 659–668.
[13] Lorenz Hübschle-Schneider and Peter Sanders. 2019. Parallel Weighted Random Sampling. In *27th European Symposium on Algorithms (ESA 2019)*. 59:1–59:24.
[14] Lorenz Hübschle-Schneider and Peter Sanders. 2019. Communication-Efficient (Weighted) Reservoir Sampling from Fully Distributed Data Streams. arXiv:1910.11069
[15] Rajesh Jayaram, Gokarna Sharma, Srikanta Tirthapura, and David P. Woodruff. 2019. Weighted Reservoir Sampling from Distributed Streams. In *38th ACM Symposium on Principles of Database Systems (PODS '19)*. 218–235.
[16] Kim-Hung Li. 1994. Reservoir-sampling algorithms of time complexity $O(n(1+\log(N/n)))$. *ACM Trans. Math. Softw.* 20, 4 (1994), 481–493.
[17] Peter Sanders, Sebastian Lamm, Lorenz Hübschle-Schneider, Emanuel Schrade, and Carsten Dachsbacher. 2018. Efficient Random Sampling – Parallel, Vectorized, Cache-Efficient, and Online. *ACM Trans. Math. Softw.* 44, 3 (2018), 29:1–29:14.
[18] Peter Sanders, Kurt Mehlhorn, M. Dietzfelbinger, and R. Dementiev. 2019. *Sequential and Parallel Algorithms and Data Structures – The Basic Toolbox*. Springer.
[19] Peter Sanders, Sebastian Schlag, and Ingo Müller. 2013. Communication Efficient Algorithms for Fundamental Big Data Problems. In *2013 IEEE International Conference on Big Data*. 15–23.
[20] Kanat Tangwongsan and Srikanta Tirthapura. 2019. Parallel Streaming Random Sampling. In *Euro-Par 2019: Parallel Processing*. Springer, 451–465.
[21] Srikanta Tirthapura and David P. Woodruff. 2011. Optimal random sampling from distributed streams revisited. In *25th International Symposium on Distributed Computing (DISC '11)*. Springer, 283–297.
[22] Jeffrey S. Vitter. 1985. Random sampling with a reservoir. *ACM Trans. Math. Softw.* 11, 1 (March 1985), 37–57.
[23] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, et al. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *24th ACM Symposium on Operating Systems Principles (SOSP '13)*. 423–438.