Communication Efficient Algorithms for Top-k Selection Problems

Lorenz Hübschle-Schneider* huebschle@kit.edu
Peter Sanders* sanders@kit.edu
Ingo Müller*† ingo.mueller@kit.edu

* Karlsruhe Institute of Technology, Karlsruhe, Germany
† SAP SE, Walldorf, Germany

Abstract

We present scalable parallel algorithms with sublinear per-processor communication volume and low latency for several fundamental problems related to finding the most relevant elements in a set, for various notions of relevance: We begin with the classical selection problem with unsorted input. We present generalizations with locally sorted inputs, dynamic content (bulk-parallel priority queues), and multiple criteria. Then we move on to finding frequent objects and top-k sum aggregation. Since it is unavoidable that the output of these algorithms might be unevenly distributed over the processors, we also explain how to redistribute this data with minimal communication.

Keywords: selection, frequent elements, sum aggregation, priority queue, sampling, branch-and-bound, data redistribution, threshold algorithm

1 Introduction

Overheads due to communication latency and bandwidth limitations of the communication networks are one of the main limiting factors for distributed computing. Parallel algorithm theory has considered the latency part of this issue since its beginning. In particular, execution times polylogarithmic in the number p of processing elements (PEs) were a main focus. Borkar [9] argues that an exascale computer could only be cost-effective if the communication capabilities (bisection width) scale highly sublinearly with the amount of computation done in the connected subsystems. Google confirms that at data center scale, the network is the most scarce resource and state that they "don't know how to build big networks that deliver lots of bandwidth" [36]. In a previous paper [34] we therefore proposed to look more intensively for algorithms that require bottleneck communication volume sublinear in the local input size. More precisely, consider an input consisting of n machine words distributed over the PEs such that each PE holds $\mathcal{O}(n/p)$ words. Then sublinear communication volume means that no PE sends or receives more than $o(n/p)$ machine words of data.

Here, we combine the latency and data volume aspects. We consider some fundamental algorithmic problems that have a large input (size n) and a relatively small output (size k). Since potentially many similar problems have to be solved, both latency and communication volume have to be very small—ideally polylogarithmic in p and the input parameters. More precisely, we consider problems

that ask for the k most "relevant" results from a large set of possibilities, and aim to obtain low bottleneck communication overhead.

In the simplest case, these are totally ordered elements and we ask for the k smallest of them—the classical selection problem. Several variants of this problem are studied in Section 4. For the classical variant with unsorted inputs, a careful analysis of a known algorithm [31] shows that the previously assumed random allocation of the inputs is actually not necessary and we get running time $\mathcal{O}(\frac{n}{p} + \log p)$. For locally sorted input we get latency $\mathcal{O}(\log^2 kp)$. Interestingly, we can return to logarithmic latency if we are willing to relax the output size k by a constant factor. This uses a new technique for obtaining a pivot element with a given rank that is much simpler than the previously proposed techniques based on sorting.

A data structure generalization of the selection problem are bulk-parallel priority queues. Previous parallel priority queues are not communication efficient in the sense that they move the elements around, either by sorting [13] or by random allocation [31]. Section 5 generalizes the results on selection from Section 4. The key to making this work is to use an appropriately augmented search tree data structure to efficiently support insertion, deletion, and all the operations needed by the parallel selection algorithm.

A prominent problem in information retrieval is to extract the k most relevant objects (e.g. documents), where relevance is determined by a monotonous function that maps several individual scoring functions to the overall relevance. For each individual score, a list of objects is precomputed that stores the objects in order of decreasing score. This is a multicriteria generalization of the sorted top-k selection problem discussed in Section 4. In Section 6 we parallelize established sequential algorithms for this problem. The single-criterion selection algorithms are used as subroutines—the sorted one for approximating the list elements scanned by the sequential algorithm and the unsorted one to actually identify the output. The algorithm has polylogarithmic overhead for coordinating the PEs and manages to contain unavoidable load imbalance in a single phase of local computation. This is achieved with a fast estimator of the output size generated with a given number of scanned objects.

A fundamental problem in data mining is finding the most frequently occurring objects. This is challenging in a distributed setting since the globally most frequent elements do not have to be locally frequent on any particular PE. In Section 7 we develop very fast sampling-based algorithms that find a (ε, δ)-approximation or *probably approximately correct answer*, i.e., with probability at least $1 - \delta$ the output is correct within εn. The algorithms run in time logarithmic in n, p, and $1/\delta$. From a simple algorithm with running time factor $1/\varepsilon^2$ we go to more sophisticated ones with factor $1/\varepsilon$. We also show how to compute the exact result with probability at least $1 - \delta$ if the elements are non-uniformly distributed.

Subsequently, we generalize these results to sum aggregation, where object occurrences are associated with a value. In Section 8, we are thus looking for the objects whose values add up to the highest sums.

All of the above algorithms have the unavoidable limitation that the output may be unevenly distributed over the PEs for general distributions of the input objects. This can lead to load imbalance affecting the efficiency of the overall application. Offsetting this imbalance will require communication so that one might argue that striving for communication efficiency in the selection process is in vain. However, our methods have several advantages over non-communication efficient selection algorithms. First of all, priorities can be ignored during redistribution since all selected elements are relevant. Hence, we can employ any data redistribution algorithm we want. In particular, we can

Table 1: Our main results. Parameters: input size n; output size k; number of PEs p; startup overhead α; communication cost per word β; relative error ε; failure probability δ. Monitoring queries are marked with †.

Problem	**Asymptotic running time in our model $\mathcal{O}(\cdot)$**	
	old	new
Unsorted Selection	$\Omega\left(\beta\frac{n}{p}\right) + \alpha \log p$ [31]	$\frac{n}{p} + \beta \min\left(\sqrt{p}\frac{\log n}{\log p}, \frac{n}{p}\right) + \alpha \log p$
Sorted Selection	$\Omega(k \log n)$ (seq.) [38]	$\alpha \log^2 kp \mid \alpha \log kp$ (flexible k)
Bulk Priority Queue insert* + deleteMin*	$\log \frac{n}{k} + \alpha\left(\frac{k}{p} + \log p\right)$ [31]	$\alpha \log^2 kp \mid \alpha \log kp$ (flexible k)
Heavy Hitters	$\frac{n}{p} + \alpha\frac{\sqrt{p}}{\varepsilon} \log n \cdot \log \frac{\log n}{\delta\varepsilon}$ [19] †	cf. top-k most frequent objects
Top-k Most Frequent Objects	$\Omega\left(\frac{n}{p} + \beta\frac{k}{\varepsilon} + \alpha\frac{1}{\varepsilon}\right)$ [3] †	$\frac{n}{p} + \beta\frac{1}{\varepsilon}\sqrt{\frac{\log p}{p} \log \frac{n}{\delta}} + \alpha \log n$
Top-k Sum Aggregation	$\frac{n}{p} + \beta\frac{1}{\varepsilon}\sqrt{p \log \frac{n}{\delta}} + \alpha p$ (centralized)	$\frac{n}{p} + \beta\frac{\log p}{\varepsilon}\sqrt{\frac{1}{p} \log \frac{n}{\delta}} + \alpha \log n$
Multicriteria Top-k	not comparable	$\log K(m^2 \log K + \beta m + \alpha \log p)$

use an adaptive algorithm that only moves data if that is really necessary. In Section 9, we give one such algorithm that combines prefix sums and merging to minimize data movement and incurs only logarithmic additional delay. Delegating data movement to the responsibility of the application also has the advantage that we can exploit properties of the application that a general top-k selection algorithm cannot. For example, multicriteria selection algorithms as discussed in Section 6 are used in search engines processing a stream of many queries. Therefore, it is enough to do load balancing over a large number of queries and we can resolve persistent hot spots by adaptive data migration or replication. Another example are branch-and-bound applications [20, 31]. By randomizing necessary data migrations, we can try to steer the system away from situations with bad data distribution (a proper analysis of such an algorithm is an open problem though).

Some of our main results are listed in Table 1. Refer to Appendix B for proofs and further discussion.

2 Preliminaries

Our input usually consists of a multiset M of $|M| = n$ objects, each represented by a single machine word. If these objects are ordered, we assume that their total ordering is unique. This is without loss of generality since we can make the value v of object x unique by replacing it by the pair (v, x) for tie breaking.

Consider p processing elements (PEs) connected by a network. PEs are numbered $1..p$, where $a..b$ is a shorthand for $\{a, \ldots, b\}$ throughout this paper. Assuming that a PE can send and receive at

most one message at a time (full-duplex, single-ported communication), sending a message of size m machine words takes time $\alpha + m\beta$. We often treat the time to initiate a connection α and to send a single machine word β as variables in asymptotic analysis. A running time of $\mathcal{O}(x + \beta y + \alpha z)$ then allows us to discuss *internal work* x, *communication volume* y, and *latency* z separately. Depending on the situation, all three aspects may be important, and combining them into a single expression for running time allows us to specify them concisely.

We present some of the algorithms using high level pseudocode in a *single program multiple data* (SPMD) style—the same algorithm runs on each PE, which perform work on their local data and communicate predominantly through collective operations. Variables are local by default. We can denote data on remote PEs using the notation $x@i$, which refers to the value of variable x on PE i. For example, $\sum_i x@i$ denotes the global sum of x over all PEs, which can be computed using a sum (all-)reduction.

Collective communication. *Broadcasting* sends a message to all PEs. *Reduction* applies an associative operation (e.g., sum, maximum, or minimum) to a vector of length m. An *all-reduction* also broadcasts this result to all PEs. A *prefix-sum* (scan) computes $\sum_{i=1}^{j} x@i$ on PE j where x is a vector of length m. The *scatter* operation distributes a message of size m to a set of y PEs such that each of them gets a piece of size m/y. Symmetrically, a *gather* operations collects y pieces of size m/y on a single PE. All of these operations can be performed in time $\mathcal{O}(\beta m + \alpha \log p)$ [5, 33]. In an all-to-all personalized communication (all-to-all for short) each PE sends one message of size m to every other PE. This can be done in time $\mathcal{O}(\beta mp + \alpha p)$ using direct point-to-point delivery or in time $\mathcal{O}(\beta mp \log p + \alpha \log p)$ using indirect delivery [21, Theorem 3.24]. The all-to-all broadcast (aka gossiping) starts with a single message of size m on each PE and ends with all PEs having all these messages. This operation works in time $\mathcal{O}(\beta mp + \alpha \log p)$.

Fast inefficient Sorting. Sorting $\mathcal{O}\left(\sqrt{p}\right)$ objects can be done in time $\mathcal{O}(\alpha \log p)$ using a brute force algorithm performing all pairwise object comparisons in parallel (e.g., [2]).

Search trees. Search trees can represent a sorted sequence of objects such that a variety of operations can be supported in logarithmic time. In particular, one can insert and remove objects and search for the next largest object given a key. The operation $T.\mathsf{split}(x)$ splits T into two search trees T_1, T_2 such that T_1 contains the objects of T with key $\leq x$ and T_2 contains the objects of T with key $> x$. Similarly, if the keys in a tree T_1 are smaller than the keys in T_2 then $\mathsf{concat}(T_1, T_2)$ returns a search tree representing the concatenation of the two sequences. If one additionally stores the size of subtrees, one can also support the select operation $T[i]$, which returns the i-th largest object in T, and operation $T.\mathsf{rank}(x)$, which returns the number of objects in T of value at most x. For an example of such a data structure see [23, Chapter 7].

Chernoff bounds. We use the following Chernoff bounds [23, 26] to bound the probability that a sum $X = X_1 + \ldots + X_n$ of n independent indicator random variables deviates substantially from its expected value $\mathbf{E}[X]$. For $0 < \varphi < 1$, we have

$$\mathbf{P}[X < (1-\varphi)\mathbf{E}[X]] \leq e^{-\frac{\varphi^2}{2}\mathbf{E}[X]} \tag{1}$$

$$\mathbf{P}[X > (1+\varphi)\mathbf{E}[X]] \leq e^{-\frac{\varphi^2}{3}\mathbf{E}[X]} \ . \tag{2}$$

Function select(s : Sequence **of** Object; $k : \mathbb{N}$) : Object
  if $k = 1$ **then return** $\min_i s[1]@i$ −− base case
  $(\ell, r) :=$ pickPivots(s)
  $a := \langle e \in s : e < \ell \rangle$
  $b := \langle e \in s : \ell \leq e \leq r \rangle$
  $c := \langle e \in s : e > r \rangle$ −− partition
  $n_a := \sum_i |a|@i$; $n_b := \sum_i |b|@i$ −− reduction
  if $n_a \geq k$ **then return** select(a, k)
  if $n_a + n_b < k$ **then return** select$(c, k - n_a - n_b)$
  return select$(b, k - n_a)$

Algorithm 1: Communication efficient selection.

Bernoulli sampling. A simple way to obtain a sample of a set M of objects is to select each object with probability ρ independent of the other objects. For small sampling probability ρ, the naive implementation can be significantly accelerated from time $\mathcal{O}(|M|)$ to expected time $\mathcal{O}(\rho\,|M|)$ by using skip values—a value of x indicates that the following $x-1$ elements are skipped and the x-th element is sampled. These skip values follow a geometric distribution with parameter ρ and can be generated in constant time.

Distributed FR-Selection [31] Floyd and Rivest [16] developed a modification of quickselect using two pivots in order to achieve a number of comparisons that is close to the lower bound. This algorithm, FR-select, can be adapted to a distributed memory parallel setting [31]. FR-select picks the pivots based on sorting a random sample S of $\mathcal{O}\left(\sqrt{p}\right)$ objects, which can easily be done in logarithmic time (e.g., the simple algorithm described in [2] performs brute-force comparisons of all possible pairs of sample elements). Pivots ℓ and r are chosen as the sample objects with ranks $k\,|S|\,/n \pm \Delta$ where $\Delta = p^{1/4+\delta}$ for some small constant $\delta > 0$. For the analysis, one can choose $\delta = 1/6$, which ensures that with high probability the range of possible values shrinks by a factor $\Theta(p^{1/3})$ in every level of recursion.

It is shown that a constant number of recursion levels suffice if $n = \mathcal{O}(p \log p)$ and if the objects are distributed randomly. Note that the latter assumption limits communication efficiency of the algorithm since it requires moving all objects to random PEs for general inputs.

Bulk Parallel Priority Queues. A natural way to parallelize priority queues is to use bulk operations. In particular, operation deleteMin* supports deletion of the k smallest objects of the queue. Such a data structure can be based on a heap where nodes store sorted sequences rather than objects [13]. However, an even faster and simpler randomized way is to use multiple sequential priority queues—one on each PE [31]. This data structure adopts the idea of Karp and Zhang [20] to give every PE a representative approximate view of the global situation by sending inserted objects to random queues. However, in contrast to Karp and Zhang [20], [31] implements an exact deleteMin* using parallel selection and a few further tricks. Note that the random insertions limit communication efficiency in this case.

3 More Related Work

The fact that the amount of communication is important for parallel computing is a widely studied issue. However, there are very few results on achieving sublinear communication volume per processor. In part, this is because models like parallel external memory or resource-oblivious models assume that the input is located in a global memory so that processing it requires loading all of it into the local caches at least once. Refer to [34] for a more detailed discussion.

A good framework for studying bottleneck communication volume is the BSP model [37] where a communication round with maximum amount of data per PE h (maximized over sent and received data) takes time $\ell + hg$ for some machine parameters ℓ (latency) and g (gap). Summing the values of h over all rounds yields the bottleneck communication volume we consider. We do not use the BSP model directly because we make heavy use of collective communication routines which are not directly represented in the BSP model. Further, the latency parameter ℓ is typically equal to αp in real-world implementations of the BSP model (see also [2]). Also, we are not aware of any concrete results in the BSP model on sublinear communication volume in general or for top-k problems in particular. A related recent line of research considers algorithms based on MapReduce (e.g., [17]). Communication volume is an important issue there but it seems difficult to achieve sublinear communication volume since the MapReduce primitive does not model locality well.

Recently, *communication avoiding algorithms* have become an important research direction for computations in high performance linear algebra and related problems, e.g. [11]. However, these results only apply to nested loops with array accesses following a data independent affine pattern and the proven bounds are most relevant for computations with work superlinear in the input size. We go into the opposite direction looking at problems with linear or even sublinear work.

Selection. Plaxton [28] shows a superlogarithmic lower bound for selection on a large class of interconnection networks. This bound does not apply to our model, since we assume a more powerful network.

Parallel Priority Queues. There has been intensive work on parallel priority queues. The most scalable solutions are our randomized priority queue [31] and Deo and Prasad's parallel heap [13]. Refer to [39] for a recent overview of further approaches, most of which are for shared memory architectures and mostly operate on centralized data structures with limited scalability.

Multicriteria, Most Frequent Objects. Considerable work has been done on distributed top-k computations in wide area networks, sensor networks and for distributed streaming models [4, 10, 19, 25, 40]. However, these papers use a master–worker approach where all communication has to go over a master node. This implies up to p times higher communication volume compared to our results. Only rarely do randomized versions with better running times exist, but nevertheless, communication volume at the master node still increases with $\sqrt{p}$ [19]. Moreover, the scalability of TPUT [10] and KLEE [25] is severely limited by the requirement that the number of workers cannot exceed the number of criteria. Furthermore, the top-k most frequent objects problem has received little attention in distributed streaming algorithms. The only work we could find requires $\mathcal{O}(Np)$ working memory at the master node, where $N = \mathcal{O}(n)$ is the number of distinct objects in the union of the streams [4]. The majority of papers instead considers the significantly easier problem of identifying the *heavy hitters*, i.e. those objects whose occurrences account for more than a fixed proportion of

the input, or *frequency tracking*, which tracks the approximate frequencies of all items, but requires an additional selection step to obtain the most frequent ones [19, 40].

Sum Aggregation Much work has been done on aggregation in parallel and distributed settings, e.g. in the database community [12, 22, 27]. However, these papers all ask for *exact* results for *all* objects, not approximations for the k most important ones. We do not believe that exact queries are can be answered in a communication-efficient manner, nor could we find any works that consider the same problem, regardless of communication efficiency.

Data redistribution using prefix sums is standard [8]. But combining it with merging allows adaptive communication volume as described in Section 9, which seems to be a new result. There are algorithms for minimizing communication cost of data distribution in arbitrary networks. However, these are much more expensive. Meyer auf der Heide et al. [24] use maximum flow computations. They also give an algorithm for meshes, which needs more steps than the lower bound though. Solving the diffusion equations minimizes the sum of the squares of the data to be moved in order to achieve perfect load balance [14]. However, solving this equation is expensive and for parallel computing we are more interested in the bottleneck communication volume.

4 Selection

We consider the problem of identifying the elements of rank $\leq k$ from a set of n objects, distributed over p processors. First, we analyze the classical problem with unsorted input, before turning to locally sorted input in Sections 4.2 and 4.3.

4.1 Unsorted Input

Our first result is that using some minor adaptations and a more careful analysis, the parallel FR-algorithm from Section 2 does not actually need randomly distributed data.

Theorem 1 *Consider n elements distributed over p PE such that each PE holds $\mathcal{O}(n/p)$ elements. The k globally smallest of these elements can be identified in expected time*

$$\mathcal{O}\left(\frac{n}{p} + \beta \min\left(\sqrt{p}\log_p n, \frac{n}{p}\right) + \alpha \log n\right) \quad .$$

Proof. (Outline) The algorithm from [31] computes the pivots based on a sample S of size $\mathcal{O}\big(\sqrt{p}\big)$. There, this is easy since the objects are randomly distributed in all levels of recursion. Here we can make no such assumption and rather assume Bernoulli sampling with probability $\sqrt{p}/\sum_i |s@i|$. Although this can be done in time proportional to the local sample size, we have to be careful since (in some level of recursion) the input might be so skewed that most samples come from the same processor. Since the result on sorting only works when the input data is uniformly distributed over the PEs, we have to account additional time for spreading the samples over the PEs.

Let x denote the PE which maximizes $|s@x|$ in the current level of recursion in the algorithm from Figure 1. The total problem size shrinks by a factor of $\Omega(p^{1/3})$ in each level of recursion with high probability. Hence, after a constant number of recursion levels, x is $\mathcal{O}(n/p)$ as well. Moreover, after $\log_{\Omega(p^{1/3})} \frac{n}{p} = \mathcal{O}(\log_p \frac{n}{p})$ further levels of recursion, the problem is solved completely with high probability. Overall, the total number of recursion levels is $\mathcal{O}(1 + \log_p \frac{n}{p}) = \mathcal{O}\big(\log_p n\big)$.

This already implies the claimed $\mathcal{O}(n/p)$ bound on internal computation – $\mathcal{O}(1)$ times $\mathcal{O}(n/p)$ until x has size $\mathcal{O}(n/p)$ and a geometrically shrinking amount of work after that.

Of course, the same bound $\mathcal{O}(\beta\frac{n}{p})$ also applies to the communication volume. However, we also know that even if all samples come from a single PE, the communication volume in a single level of recursion is $\mathcal{O}\big(\sqrt{p}\big)$.

Finally, the number of startups can be limited to $\mathcal{O}(\log p)$ per level of recursion by using a collective scatter operation on those PEs that have more than one sample object. This yields $\mathcal{O}\big(\log p \log_p n\big) = \mathcal{O}(\log n)$. □

Corollary 2 *If α and β are viewed as constants, the bound from Theorem 1 reduces to $\mathcal{O}\Big(\frac{n}{p} + \log p\Big)$.*

The bound from Theorem 1 immediately reduces to $\mathcal{O}\Big(\frac{n}{p} + \log n\Big)$ for constant α and β. We can further simplify this by observing that when the $\log n$ term dominates n/p, then $\log n = \mathcal{O}(\log p)$.

4.2 Locally Sorted Input

Selection on locally sorted input is easier than the unsorted problem from Section 4.1 since we only have to consider the locally smallest objects and are able to locate keys in logarithmic time. Indeed, this problem has been studied as the *multisequence selection* problem [35, 38].

In [2], we propose a particularly simple and intuitive method based on an adaptation of the well-known quickselect algorithm [18, 23]. For self-containedness, we also give that algorithm in Appendix A, adapting it to the need in the present paper. This algorithm needs running time $\mathcal{O}\big(\alpha \log^2 kp\big)$ and has been on the slides of Sanders' lecture on parallel algorithms since 2008 [32]. Algorithm 9 can be viewed as a step backwards compared to our algorithm from Section 4.1 as using a single random pivot forces us to do a deeper recursion. We do that because it makes it easy to tolerate unbalanced input sizes in the deeper recursion levels. However, it is possible to reduce the recursion depth to some extent by choosing pivots more carefully and by using multiple pivots at once. We study this below for a variant of the selection problem where we are flexible about the number k of objects to be selected.

4.3 Flexible k, Locally Sorted Input

The $\mathcal{O}\big(\log^2 pk\big)$ startups incurred in Section 4.2 can be reduced to $\mathcal{O}(\log pk)$ if we are willing to give up some control over the number of objects that are actually returned. We now give two input parameters $\underline{k}$ and $\overline{k}$ and the algorithm returns the k smallest objects so that $\underline{k} \le k \le \overline{k}$. We begin with a simple algorithm that runs in logarithmic time if $\overline{k} - \underline{k} = \Omega\left(\overline{k}\right)$ and then explain how to refine that for the case $\overline{k} - \underline{k} = o(\overline{k})$.

The basic idea for the simple algorithm is to take a Bernoulli sample of the input using a success probability of $1/x$, for $x \in \underline{k}..\overline{k}$. Then, the expected rank of the smallest sample object is x, i.e., we have a truthful estimator for an object with the desired rank. Moreover, this object can be computed efficiently if working with locally sorted data: the local rank of the smallest local sample is geometrically distributed with parameter $1/x$. Such a number can be generated in constant time. By computing the global minimum of these locally smallest samples, we can get the globally smallest sample v in time $\mathcal{O}(\alpha \log p)$. We can also count the exact number k of input objects bounded by this estimate in time $\mathcal{O}(\log k + \alpha \log p)$—we locate v in each local data set in time $\mathcal{O}(\log k)$ and then

sum the found positions. If $k \in \underline{k}..\overline{k}$, we are done. Otherwise, we can use the acquired information as in any variant of quickselect.

At least in the recursive calls, it can happen that $\overline{k}$ is close to the total input size n. Then it is a better strategy to use a dual algorithm based on computing a global maximum of a Bernoulli sample. In Algorithm 2 we give pseudocode for a combined algorithm dynamically choosing between these two cases. It is an interesting problem which value should be chosen for x. The formula used in Algorithm 2 maximizes the probability that $k \in \underline{k}..\overline{k}$. This value is close to the arithmetic mean when $\overline{k}/\underline{k} \approx 1$ but it is significantly smaller otherwise. The reason for this asymmetry is that larger sampling rates decrease the variance of the geometric distribution.

Theorem 3 *If $\overline{k} - \underline{k} = \Omega\left(\overline{k}\right)$, then Algorithm* amsSelect *from Figure 2 finds the k smallest elements with $k \in \underline{k}..\overline{k}$ in expected time $\mathcal{O}\big(\log \overline{k} + \alpha \log p\big)$.*

Proof. (Outline) One level of recursion takes time $\mathcal{O}(\alpha \log p)$ for collective communication operations (min, max, or sum reduction) and time $\mathcal{O}\big(\log \overline{k}\big)$ for locating the pivot v. It remains to show that the expected recursion depth is constant.

We actually analyze a weaker algorithm that keeps retrying with the same parameters rather than using recursion and that uses probe $x = \underline{k}$. We show that, nevertheless, there is a constant success probability (i.e., $\underline{k} \le k \le \overline{k}$ with constant probability). The rank of v is geometrically distributed with parameter $1/x$. The success probability becomes

$$\sum_{k=\underline{k}}^{\overline{k}} \left(1 - \frac{1}{\underline{k}}\right)^{k-1} \frac{1}{\underline{k}} = \frac{\underline{k}}{\underline{k}-1}\left(1 - \frac{1}{\underline{k}}\right)^{\underline{k}} - \left(1 - \frac{1}{\underline{k}}\right)^{\overline{k}} \approx \frac{1}{e} - e^{-\frac{\overline{k}}{\underline{k}}}$$

which is a positive constant if $\overline{k} - \underline{k} = \Omega\left(\overline{k}\right)$. □

Multiple Concurrent Trials. The running time of Algorithm 2 is dominated by the logarithmic number of startup overheads for the two reduction operations it uses. We can exploit that reductions can process long vectors using little additional time. The idea is to take d Bernoulli samples of the input and to compute d estimates for an object of rank x. If any of these estimates turns out to have exact rank between $\underline{k}$ and $\overline{k}$, the recursion can be stopped. Otherwise, we solve a recursive instance consisting of those objects enclosed by the largest underestimate and the smallest overestimate found.

Theorem 4 *If $\overline{k} - \underline{k} = \Omega\left(\overline{k}/d\right)$, an algorithm processing batches of d Bernoulli samples can be implemented to run in expected time $\mathcal{O}\big(d \log \overline{k} + \beta d + \alpha \log p\big)$.*

Proof. (Outline) A single level of recursion runs in time $\mathcal{O}\big(d \log \overline{k} + \beta d + \alpha \log p\big)$. Analogous to the proof of Theorem 3, it can be shown that the success probability is $\Omega\left(1/d\right)$ for a single sample. This implies that the probability that any of the d independent samples is successful is constant. □

For example, setting $d = \Theta(\log p)$, we obtain communication time $\mathcal{O}(\alpha \log p)$ and $\mathcal{O}(\log k \log p)$ time for internal work.

Function amsSelect(s : Object[]; $\underline{k}$: $\mathbb{N}$; $\overline{k}$: $\mathbb{N}$) : Object[]

if $\underline{k} < n - \overline{k}$ **then** −− min-based estimator

x:= geometricRandomDeviate $\left(1 - \left(\frac{\underline{k}-1}{\overline{k}}\right)^{\frac{1}{\overline{k}-\underline{k}+1}}\right)$

if $x > |s|$ **then** v:= ∞ **else** v:= $s[x]$

v:= $\min_{1 \le i \le p} v@i$ −− minimum reduction

else −− max-based estimator

x:= geometricRandomDeviate $\left(1 - \left(\frac{n-\overline{k}}{n-\underline{k}+1}\right)^{\frac{1}{\overline{k}-\underline{k}+1}}\right)$

if $x > |s|$ **then** v:= $-\infty$ **else** v:= $s[|s| - x + 1]$

v:= $\max_{1 \le i \le p} v@i$ −− maximum reduction

find j such that $s[1..j] \le v$ and $s[j+1..] > v$

k:= $\sum_{1 \le i \le p} |j@i|$ −− sum all-reduction

if $k < \underline{k}$ **then**

return $s[1..j] \cup$ amsSelect$(s[j+1..|s|], \underline{k} - k, \overline{k} - k, n - k)$

if $k > \overline{k}$ **then**

return amsSelect$(s[1..j], \underline{k}, \overline{k}, k)$

return $s[1..j]$

Algorithm 2: Approximate multisequence selection. Select the k globally smallest out of n objects with $\underline{k} \le k \le \overline{k}$. Function geometricRandomDeviate is a standard library function for generating geometrically distributed random variables in constant time [29].

5 Bulk Parallel Priority Queues

We build a global bulk-parallel priority queue from local sequential priority queues as in [20, 31], but never actually move elements. This immediately implies that insertions simply go to the local queue and thus require only $\mathcal{O}(\log n)$ time without any communication. Of course, this complicates operation deleteMin*. The number of elements to be retrieved from the individual local queues can vary arbitrarily and the set of elements stored locally is not at all representative for the global content of the queue. We can not even afford to individually remove the objects output by deleteMin* from their local queues.

We therefore replace the ordinary priority queues used in [31] by search tree data structures that support insertion, deletion, selection, ranking, splitting and concatenation of objects in logarithmic time (see also Section 2). To become independent of the actual tree size of up to n, we furthermore augment the trees with two arrays storing the path to the smallest and largest object respectively. This way, all required operations can be implemented to run in time $\mathcal{O}(\log k)$ rather than $\mathcal{O}(\log n)$.

Operation deleteMin* now becomes very similar to the multi-sequence selection algorithms from Section 4.3 and Appendix A. The only difference is that instead of sorted arrays, we are now working on search trees. This implies that selecting a local object with specified local rank (during sampling) now takes time $\mathcal{O}(\log k)$ rather than constant time. However, asymptotically, this makes no difference since for any such selection step, we also perform a ranking step, which takes time $\mathcal{O}(\log k)$ anyway in both representations.

One way to implement the recursion in the selection algorithms is via splitting. Since the split operation is destructive, after returning from the recursion, we have to reassemble the previous state using concatenation. Another way that might be faster and simpler in practice is to represent a subsequence of s by s itself plus cursor information specifying rank and key of the first and last object of the subsequence.

Now, we obtain the following by applying our results on selection from Section 4.

Theorem 5 *Operation* deleteMin* *can be implemented to run in the following expected times. With fixed batch size k, expected time $\mathcal{O}(\alpha \log^2 kp)$ suffices. For flexible batch size in $\underline{k}..\overline{k}$, where $\overline{k} - \underline{k} = \Omega\left(\overline{k}\right)$, we need expected time $\mathcal{O}(\alpha \log kp)$. If $\overline{k} - \underline{k} = \Omega\left(\overline{k}/d\right)$, expected time $\mathcal{O}\left(d \log \overline{k} + \beta d + \alpha \log p\right)$ is sufficient.*

Note that flexible batch sizes might be adequate for many applications. For example, the parallel branch-and-bound algorithm from [31] can easily be adapted: In iteration i of its main loop, it deletes the smallest $k_i = \mathcal{O}(p)$ elements (tree nodes) from the queue, expands these nodes in parallel, and inserts newly generated elements (child nodes of the processed nodes). Let $K = \sum_i k_i$ denote the total number of nodes expanded by the parallel algorithm. One can easily generalize the proof from [31] to show that $K = m + \mathcal{O}(hp)$ where m is the number of nodes expanded by a sequential best first algorithm and h is the length of the path from the root to the optimal solution. Also note that a typical branch-and-bound computation will insert significantly more nodes than it removes—the remaining queue is discarded after the optimal solutions are found. Hence, the local insertions of our communication efficient queue are a big advantage over previous algorithms, which move all nodes [20, 31].

6 Multicriteria Top-k

In the sequential setting, we consider the following problem: Consider m lists L_i of scores that are sorted in decreasing order. Overall relevance of an object is determined by a scoring function $t(x_1, \ldots, x_m)$ that is monotonous in all its parameters. For example, there could be m keywords for a disjunctive query to a fulltext search engine, and for each pair of a keyword and an object, it is known how relevant this keyword is for this object. Many algorithms used for this setting are based on Fagin's threshold algorithm [15]. The lists are partially scanned and the algorithm maintains lower bounds for the relevance of the scanned objects as well as upper bounds for the unscanned objects. Bounds for a scanned object x can be tightened by retrieving a score value for a dimension for which x has not been scanned yet. These *random accesses* are more expensive than scanning, in particular if the lists are stored on disk. Once the top-k scanned objects have better lower bounds than the best upper bound of an unscanned object, no more scanning is necessary. For determining the exact relevance of the top-k scanned objects, further random accesses may be required. Various strategies for scanning and random access yield a variety of variants of the threshold algorithm [6].

The original threshold algorithm works as follows [15]: In each of K iterations of the main loop, scan one object from each list and determine its exact score using random accesses. Let x_i denote the smallest score of a scanned object in L_i. Once at least k scanned objects have score at least $t(x_1, \ldots, x_m)$, stop, and output them.

We consider a distributed setting where each PE has a subset of the objects and m sorted lists ranking its locally present objects. We describe communication efficient distributed algorithms that approximate the original threshold algorithm (TA) [15]. First, we give a simple algorithm assuming

Procedure DTA($\langle L_1, \ldots, L_m \rangle, t, k$)
  $K := \left\lceil \frac{k}{mp} \right\rceil$ −− lower bound for K
  do
    for $i := 1$ **to** m **do**
      $L'_i := \mathsf{amsSelect}(L_i, K, 2K, |L_i|)$, $(\cdot, x_i) := L'_i.\text{last}$ −− x_i min selected score
    $t_{\min} := t(x_1, \ldots, x_m)$ −− threshold
    for $i := 1$ **to** m −− each PE locally
      $R_i, H_i := 0$
      for $j := 1$ **to** $y = \mathcal{O}(\log K)$
        sample an element $x = (o, s)$ from L'_i
        if $\exists_{j<i, s'} : (o, s') \in L'_j$ **then** $R_i{+}{+}$ −− ignore samples contained in other L'_j
        elsif $t(o) \geq t_{\min}$ **then** $H_i{+}{+}$ −− $t(o)$ does lookups in all L_i to compute score
      $\ell_i := |L'_i| \left(1 - \frac{R_i}{y}\right) \cdot \frac{H_i}{y}$ −− truthful estimator of #hits for L'_i on this PE
    $H := \sum_{j=1}^{p} \left(\sum_{i=1}^{m} \ell_i\right) @j$
    $K := 2K$ −− exponential search
  until $H \geq 2k$ −− $\Rightarrow \geq k$ hits found whp
  return $(t_{\min}, \langle L'_1, \ldots, L'_m \rangle)$

Algorithm 3: Multicriteria Top-k for arbitrary data distribution (DTA) on Lists $L_1, \ldots, L_m$ with scoring function t

random data distribution of the input objects (RDTA) and then describe a more complex algorithm for arbitrary data distribution (DTA).

Random Data Distribution. Since the data placement is independent of the relevance of the objects, the top-k objects are randomly distributed over the PEs. Well known balls-into-bins bounds give us tight high probability bounds on the maximal number $\overline{k}$ of top-k objects on each PE [30]. Here, we work with the simple bound $\overline{k} = \mathcal{O}(\frac{k}{p} + \log p)$. The RDTA algorithm simply runs TA locally to retrieve the $\overline{k}$ locally most relevant objects on each PE as result candidates. It then computes a global threshold as the maximum of the local thresholds and verifies whether at least k candidate objects are above this global threshold. In the positive case, the k most relevant candidates are found using the selection algorithm from [31]. Otherwise, $\overline{k}$ is increased and the algorithm is restarted. In these subsequent iterations, PEs whose local threshold is worse than the relevance of the k-th best element seen so far do not need to continue scanning. Hence, we can also trade less local work for more rounds of communication by deliberately working with an insufficiently small value of $\overline{k}$.

Arbitrary Data Distribution. We give pseudocode for Algorithm DTA in Figure 3.

Theorem 6 *Algorithm DTA requires expected time* $\mathcal{O}\big(m^2 \log^2 K + \beta m \log K + \alpha \log p \log K\big)$ *to identify a set of at most* $\mathcal{O}(K)$ *objects that contains the set of* K *objects scanned by TA.*

Proof. (Outline) Algorithm DTA "guesses" the number K of list rows scanned by TA using exponential search. This yields $\mathcal{O}(\log K)$ *rounds* of DTA. In each round, the approximate multisequence

selection algorithm from Section 4.3 is used to approximate the globally K-th largest score x_i in each list. Define L'_i as the prefix of L_i containing all objects of score at least x_i. This takes time $\mathcal{O}(\log K + \alpha \log p)$ in expectation by Theorem 3. Accumulating the searches for all m lists yields $\mathcal{O}(m \log K + \beta m + \alpha \log p)$ time. Call an object selected by the selection algorithm a *hit* if its relevance is above the threshold $t(x_1, \ldots, x_m)$. DTA estimates the number of hits using sampling. For each PE and list i separately, $y = \mathcal{O}(\log K)$ objects are sampled from L'_i. Each sample takes time $\mathcal{O}(m)$ for evaluating $t(\cdot)$. Multiplying this with m lists and $\log K$ iterations gives a work bound of $m^2 \log K$. Note that this is sublinear in the work done by the sequential algorithm which takes time $m^2 K$ to scan the K most important objects in every list. To eliminate bias for objects selected in multiple lists L'_j, DTA only counts an object x if it is sampled in the first list that contains it. Otherwise, x is rejected. DTA also counts the number of rejected samples R. Let H denote the number of nonrejected hits in the sample. Then, $\ell'_i := |L'_i|\,(1 - R/y)$ is a truthful estimate for the length of the list with eliminated duplicates and $\frac{H}{y}\ell'_i$ is a truthful estimate for the number of hits for the considered list and PE. Summing these mp values using a reduction operation yields a truthful estimate for the overall number of hits. DTA stops once this estimate is large enough such that with high probability the actual number of hits is at least k. The output of DTA are the prefixes $L'_1, \ldots, L'_m$ of the lists on each PE, the union of which contains the k most relevant objects with high probability. It also outputs the threshold $t(x_1, \ldots, x_m)$. In total, all of this combined takes expected time $\mathcal{O}(m^2 \log^2 K + \beta m \log K + \alpha \log p \log K)$. □

Actually computing the k most frequent objects amounts to scanning the result lists to find all hits and, if desired, running a selection algorithm to identify the k most relevant among them. The scanning step may involve some load imbalance, as in the worst case, all hits are concentrated on a single PE. This seems unavoidable unless one requires random distribution of the objects. However, in practice it may be possible to equalize the imbalance over a large number of concurrently executed queries.

Refinements. We can further reduce the latency of DTA by trying several values of K in each iteration of algorithm DTA. Since this involves access to only few objects, the overhead in internal work will be limited. In practice, it may also be possible to make good guesses on K based on previous executions of the algorithm.

7 Top-k Most Frequent Objects

We describe two *probably approximately correct (PAC)* algorithms to compute the top-k most frequent objects of a multiset M with $|M| = n$, followed by a *probably exactly correct (PEC)* algorithm for suitable inputs. Sublinear communication is achieved by transmitting only a small random sample of the input. For bound readability, we assume that M is distributed over the p PEs so that none has more than $\mathcal{O}(n/p)$ objects. This is not restricting, as the algorithms' running times scale linearly with the maximum fraction of the input concentrated at one PE.

We express the algorithms' error relative to the total input size. This is reasonable—consider a large input where k objects occur twice, and all others once. If we expressed the error relative to the objects' frequencies, this input would be infeasible without communicating all elements. Thus, we refer to the algorithms' relative error as $\tilde{\varepsilon}$, defined so that the absolute error $\tilde{\varepsilon} n$ is the count of the most frequent object that was not output minus that of the least frequent object that was output,

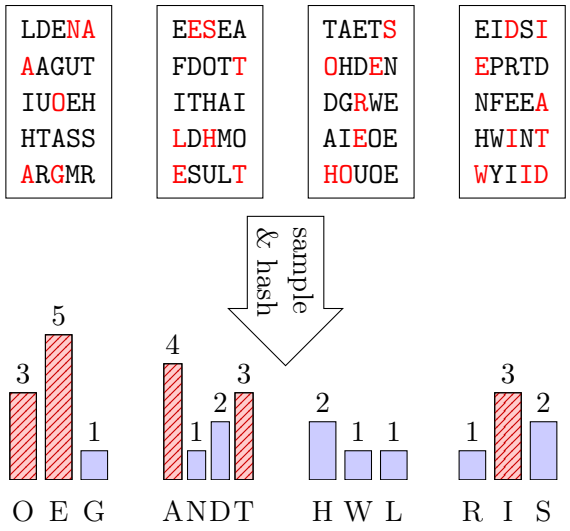


Figure 4: Example for the PAC top-k most frequent objects algorithm from Section 7.1. Top: input distributed over 4 PEs, sampled elements ($\rho = 0.3$) highlighted in red. Bottom: distributed hash table counting the samples. The $k = 5$ most frequent objects, highlighted in red, are to be returned after scaling with $1/\rho$. Estimated result: $(E, 17), (A, 13), (T, 10), (I, 10), (O, 10)$. Exact result: $(E, 16), (A, 10), (T, 10), (I, 9), (D, 8)$. Object D was missed, instead O (count 7) was returned. Thus, the algorithm's error is $8 - 7 = 1$ here.

or 0 if the result was exact. Let δ limit the probability that the algorithms exceeds bound ε, i.e. $\mathbf{P}[\tilde{\varepsilon} > \varepsilon] < \delta$. We refer to the result as an (ε, δ)-approximation.

7.1 Basic Approximation Algorithm

First, we take a Bernoulli sample of the input. Sampling is done locally. The frequencies of the sampled objects are counted using distributed hashing—a local object count with key k is sent to PE $h(k)$ for a hash function h that we here expect to behave like a random function. We then select the k most frequently sampled objects using the unsorted selection algorithm from Section 4.1. An example is illustrated in Figure 4.

Theorem 7 *Algorithm* PAC *can be implemented to compute an (ε, δ)-approximation of the top-k most frequent objects in expected time $\mathcal{O}(\beta \frac{\log p}{p\varepsilon^2} \log \frac{k}{\delta} + \alpha \log n)$.*

Lemma 8 *For sampling probability ρ, Algorithm* PAC *runs in $\mathcal{O}\left(\frac{n\rho}{p} + \beta \frac{n\rho}{p} \log p + \alpha \log n\right)$ time in expectation.*

Proof. (Outline) Bernoulli sampling is done in expected time $\mathcal{O}(n\rho/p)$ by generating skip values with a geometric distribution using success probability ρ. Since the number of sample elements in a Bernoulli sample is a random variable, so is the running time. To count the sampled objects, each PE aggregates its local samples in a hash table, counting the occurrence of each sample object during the sampling process. It inserts its share of the sample into a distributed hash table [34] whose hash function we assume to behave like a random function, thus distributing the objects randomly among

the PEs. The elements are communicated using indirect delivery to maintain logarithmic latency. This requires $\mathcal{O}(\beta\frac{n\rho}{p}\log p + \alpha\log p)$ time in expectation. To minimize worst-case communication volume, the incoming sample counts are merged with a hash table in each step of the reduction. Thus, each PE receives at most one message per object assigned to it by the hash function.

From this hash table, we select the object with rank k using Algorithm 1 in expected time $\mathcal{O}(\frac{n\rho}{p} + \beta\frac{n\rho}{p} + \alpha\log n\rho)$. This pivot is broadcast to all PEs, which then determine their subset of at least as frequent sample objects in expected time $\mathcal{O}(\frac{n\rho}{p} + \alpha\log p)$. These elements are returned. Overall, the claimed time complexity follows using the estimate $p \leq n$ and $n\rho \leq n$ in order to simplify the α-term. □

Lemma 9 *Let $\rho \in (0,1)$ be the sampling probability. Then, Algorithm* PAC*'s error $\tilde{\varepsilon}$ exceeds a value of Δ/n with probability*

$$\mathbf{P}\left[\tilde{\varepsilon}n \geq \Delta\right] \leq ne^{-\frac{\Delta^2\rho k}{12n}} + ke^{-\frac{\Delta^2\rho}{8n}}.$$

Proof. We bound the error probability as follows: the probability that the error $\tilde{\varepsilon}$ exceeds some value Δ/n is at most n times the probability that a single object's value estimate deviates from its true value by more than $\Delta/2$ in either direction. For non-top-k objects, our concern is the possibility of *overestimation*, while for top-k objects, *underestimation* is interesting. These probabilities can be bounded using Chernoff bounds. We denote the count of element j in the input by x_j and in the sample by s_j. Further, let F_j be the probability that the error for element j exceeds $\Delta/2$ in either direction. Let $j \geq k$, and observe that $x_j \leq n/k$. Using Equations 1 and 2 with $X = s_j$, $\mathbf{E}[X] = \rho x_j$, and $\varphi = \frac{\Delta}{2x_j}$, we obtain:

$$F_j \leq \mathbf{P}\left[s_j \geq \rho\left(x_j + \frac{\Delta}{2}\right)\right] \leq e^{-\frac{\Delta^2\rho k}{12n}}$$

This leaves us with the most frequent $k-1$ elements, whose counts can be bounded as $x_j \leq n$. As overestimating them is not a concern, we apply the Chernoff bound in Equation 1 and obtain $\mathbf{P}\left[s_j \leq \rho\left(x_j - \frac{\Delta}{2}\right)\right] \leq e^{-\frac{\Delta^2\rho}{8n}}$. In sum, all error probabilities add up to the claimed value. □

We bound this result by an error probability δ. This allows us to calculate the minimum required sample size given δ and $\Delta = \varepsilon n$. Solving the above equation for ρ yields

$$\rho n \geq \frac{4}{\varepsilon^2} \cdot \max\left(\frac{3}{k}\ln\frac{2n}{\delta},\ 2\ln\frac{2k}{\delta}\right), \tag{3}$$

which is dominated by the latter term in most cases and yields $\rho n \geq \frac{8}{\varepsilon^2}\ln\frac{2k}{\delta}$ for the expected sample size.

Proof. *(Theorem 7)* Equation 3 yields $\rho n = \mathcal{O}\left(\frac{1}{\varepsilon^2}\log\frac{k}{\delta}\right)$. The claimed running time bound then follows from Lemma 8. □

Note that if we can afford to aggregate the local input, we can also use the Sum Aggregation algorithm from Section 8 and associate a value of 1 with each object.

7.2 Increasing Communication Efficiency

Sample sizes proportional to $1/\varepsilon^2$ quickly become unacceptably large as ε decreases. To remedy this, we iterate over the local input a second time and count the most frequently sampled objects' occurrences exactly. This allows us to reduce the sample size and improve communication efficiency at the cost of increased local computation. We call this *Algorithm EC* for *exact counting*. Again, we begin by taking a Bernoulli sample. Then we find the $k^* \geq k$ globally most frequent objects in the sample using the unsorted selection algorithm from Section 4.1, and count their frequency in the overall input exactly. The identity of these objects is broadcast to all PEs using an all-gather (gossiping, all-to-all broadcast) collective communication operation. After local counting, a global reduction sums up the local counts to exact global values. The k most frequent of these are then returned.

Lemma 10 *When counting the k' most frequently sampled objects' occurrences exactly, a sample size of $n\rho = \frac{2}{\varepsilon^2 k'} \ln \frac{n}{\delta}$ suffices to ensure that the result of Algorithm* EC *is an (ε, δ)-approximation of the top-k most frequent objects.*

Proof. With the given error bounds, we can derive the required sampling probability ρ similar to Lemma 9. However, we need not consider overestimation of the k^* most frequent objects, as their counts are known exactly. We can also allow the full margin of error towards underestimating their frequency (Δ instead of $\Delta/2$) and can ignore overestimation. This way, we obtain a total expected sample size of $\rho n \geq \frac{2}{\varepsilon^2 k^*} \ln \frac{n}{\delta}$. □

We can now calculate the value of k' that minimizes the total communication volume and obtain $k^* = \max(k, \frac{1}{\varepsilon}\sqrt{\frac{2 \log p}{p} \ln \frac{n}{\delta}})$. Substituting this into the sample size equation of Lemma 12 and adapting the running time bound of Lemma 8 then yields the following theorem.

Theorem 11 *Algorithm* EC *can be implemented to compute an (ε, δ)-approximation of the top-k most frequent objects in $\mathcal{O}\left(\frac{n}{p} + \beta \frac{1}{\varepsilon}\sqrt{\frac{\log p}{p} \cdot \log \frac{n}{\delta}} + \alpha \log n\right)$ time in expectation.*

Proof. Sampling and hashing are done as in Algorithm *PAC* (Section 7.1). We select the object with rank $k^* = \max(k, \frac{1}{\varepsilon}\sqrt{\frac{2 \log p}{p} \ln \frac{n}{\delta}})$ as a pivot. This requires expected time $\mathcal{O}(\frac{n\rho}{p} + \beta \frac{n\rho}{p} + \alpha \log n\rho)$ using Algorithm 1. The pivot is broadcast to all PEs, which then determine their subset of at least as frequent sample objects using $\mathcal{O}(\frac{n\rho}{p} + \alpha \log p)$ time in expectation. Next, these k^* most frequently sampled objects are distributed to all PEs using an all-gather operation in time $\mathcal{O}(\beta k^* + \alpha \log p)$. Now, the PEs count the received objects' occurrences in their local input, which takes $\mathcal{O}(n/p)$ time. These counts are summed up using a vector-valued reduction, again requiring $\mathcal{O}(\beta k^* + \alpha \log p)$ time. We then apply Algorithm 1 a second time to determine the k most frequent of these objects. Overall, the claimed time complexity follows by substituting k^* for k' in the sampling probability from Lemma 10. □

Substituting k^* from before then yields a total required sample size of $\rho n = \frac{1}{\varepsilon} \cdot \sqrt{\frac{p}{\log p} \ln \frac{n}{\delta}}$ for Algorithm *EC*. Note that this term grows with $\frac{1}{\varepsilon}$ instead of $1/\varepsilon^2$, reducing per-PE communication volume to $\mathcal{O}\left(\frac{1}{\varepsilon}\sqrt{\frac{\log p}{p} \log \frac{n}{\delta}}\right)$ words.

To continue the example from Figure 4, we may set $k^* = 8$. Then, the k^* most frequently sampled objects (E, A, T, I, O, D, H, S) with $(16, 10, 10, 9, 7, 8, 7, 6)$ occurrences, respectively, will be counted exactly. The result would now be correct.

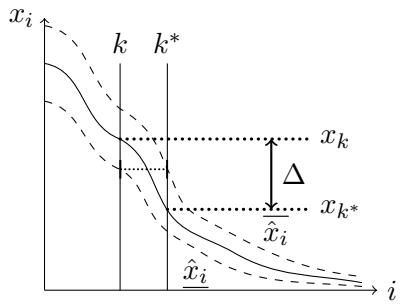


Figure 5: Example of a distribution with a gap. The dashed lines indicate the upper and lower high-probability bounds on x_i estimated from the sample

Note that Algorithm *EC* can be *less* communication efficient than Algorithm *PAC* if ε is large, i.e. the result is very approximate. Then, k^* can be prohibitively large, and the necessity to communicate the identity of the objects to be counted exactly, requiring time $\mathcal{O}(\beta k^* + \alpha \log p)$, can cause a loss in communication efficiency.

7.3 Probably Exactly Correct Algorithm

If any significant *gap* exists in the frequency distribution of the objects (see Figure 5 for an example), we perform exact counting on all likely relevant objects, determined from their sample count. Thus, choose k^* to ensure that the top-k most frequent objects in the input are among the top-k^* most frequent objects in the sample with probability at least $1 - \delta$.

A *probably exactly correct (PEC)* algorithm to compute the top-k most frequent objects of a multiset whose frequency distribution is sufficiently sloped can therefore be formulated as follows. Take a small sample with sampling probability ρ_0, the value of which we will consider later. From this small sample, we deduce the required value of k^* to fulfill the above requirements. Now, we apply Algorithm *EC* using this value of k^*. Let x_i be the object of rank i in the input, and $\hat{s}_j$ that of rank j in the small sample.

Lemma 12 *It suffices to choose k^* in such a way that $\hat{s}_{k^*} \leq \rho_0 x_k - \sqrt{2\rho_0 x_k \ln \frac{k}{\delta}} = \mathbf{E}[\hat{s}_k] - \sqrt{2\mathbf{E}[\hat{s}_k] \ln \frac{k}{\delta}}$ to ensure correctness of Algorithm* PEC*'s result with probability at least $1 - \delta$.*

Proof. We use the Chernoff bound from Equation 1 to bound the probability of a top-k object not being among the top-k^* objects of the sample. Define $\hat{s}(x_i)$ as the number of samples of the object with input rank i in the first sample, and $x(\hat{s}_j)$ to be the exact number of occurrences in the input of the object with rank j in the first sample. Using $X = \hat{s}(x_k)$, $\mathbf{E}[X] = \rho_0 x_k$, and $\varphi = 1 - \frac{\hat{s}_{k^*}}{\rho_0 x_k}$, we obtain

$$\sum_{j=1}^{k} \mathbf{P}[\hat{s}(x_j) \leq \hat{s}_{k^*}] \leq k \cdot \mathbf{P}[\hat{s}(x_k) \leq \hat{s}_{k^*}]$$
$$\leq k \cdot e^{-\left(1 - \frac{\hat{s}_{k^*}}{\rho_0 x_k}\right)^2 \frac{\rho_0 x_k}{2}} .$$

We bound this value by the algorithm's error probability δ and solve for $\hat{s}_{k^*}$, which yields the claimed value. □

This only works for sufficiently sloped distributions, as otherwise $k^* \gg k$ would be necessary. Furthermore, it is clear that the choice of ρ_0 presents a trade–off between the time and communication spent on the first sample and the exactness of k^*, which we have to estimate more conservatively if the first sample is small. This is due to less precise estimations of $\mathbf{E}[\hat{s}_k] = \rho_0 x_k$ if ρ_0 is small. To keep things simple, we can choose a relative error bound ε_0 and use the sample size from our PAC algorithm of Theorem 7. The value of ε_0—and thus ρ_0—that minimizes the communication volume depends on the distribution of the input data.

Theorem 13 *If the value of k^* computed from Lemma 12 satisfies $k^* = \mathcal{O}(k)$, then Algorithm* PEC *requires time asymptotically equal to the sum of the running times of algorithms* PAC *and* EC *from Theorems 7 and 11.*

Proof. *(Outline)* In the first sampling step, we are free to choose an arbitrary relative error tolerance ε_0. The running time of this stage is $\mathcal{O}(\beta \frac{\log p}{p\varepsilon_0^2} \log \frac{n}{\delta} + \alpha \log n)$ by Theorem 7. We then estimate k^* by substituting the high-probability bound $\mathbf{E}[\hat{s}_k] \geq \hat{s}_k - \sqrt{2\hat{s}_k \ln(1/\delta)}$ (whp) for its expected value in Lemma 12 (note that as $\hat{s}_k$ increases with growing sample size and thus grows with falling ε_0, the precision of this bound increases). In the second stage, we can calculate the required value of ε from k^* by solving the expression for k^* in the proof of Theorem 11 for ε, and obtain $\varepsilon = \frac{1}{k^*}\sqrt{\frac{2\log p}{p} \log \frac{n}{\delta}}$. Since $k^* = \mathcal{O}(k)$, the second stage's running time is as in Theorem 11. In sum, the algorithm requires the claimed running time. □

Zipf's Law In its simplest form, Zipf's Law states that the frequency of an object from a multiset M with $|M| = n$ is inversely proportional to its rank among the objects ordered by frequency. Here, we consider the general case with exponent parameter s, i.e. $x_i = n\frac{i^{-s}}{H_{n,s}}$, where $H_{n,s} = \sum_{i=1}^{n} i^{-s}$ is the n-th *generalized harmonic number*.

Theorem 14 *For inputs distributed according to Zipf's law with exponent s, a sample size of $\rho n = 4k^s H_{n,s} \ln \frac{k}{\delta}$ is sufficient to compute a probably exactly correct result. Algorithm* PEC *then requires expected time $\mathcal{O}\left(\frac{n}{p} + \beta\frac{\log p}{p} k^s H_{n,s} \log \frac{k}{\delta} + \alpha \log n\right)$ to compute the k most frequent objects with probability at least $1 - \delta$.*

Proof. Knowing the value distribution, we do not need to take the first sample. Instead, we can calculate the expected value of k^* directly from the proof of Lemma 12 and obtain

$$\mathbf{E}[k^*] = k\left(1 - \sqrt{\frac{2\ln\frac{k}{\delta}}{\rho x_k}}\right)^{-\frac{1}{s}}.$$

This immediately yields $\rho > \frac{2}{x_k} \ln \frac{k}{\delta}$, and we choose $\rho = \frac{4}{x_k} \ln \frac{k}{\delta} = 4\frac{k^s H_{n,s}}{n} \ln \frac{k}{\delta}$, i.e. twice the required minimum value. This gives us the claimed sample size. Now, we obtain $\mathbf{E}[k^*] = k\left(2 + \sqrt{2}\right)^{\frac{1}{s}} \approx \sqrt[s]{3.41}k$. In particular, k^* is only a constant factor away from k. Plugging the above sampling

probability into the running time formula for our algorithm using exact counting, we obtain the exact top-k most frequent objects with probability at least $1 - \delta$ in the claimed running time. □

Note that the number of frequent objects decreases sharply with $s > 1$, as the k-th most frequent one has a relative frequency of only $\mathcal{O}(k^{-s})$. The $k^s H_{n,s}$ term in the communication volume is thus small in practice, and, in fact, unavoidable in a sampling-based algorithm. (One can easily verify this claim by observing that this factor is the reciprocal of the k-th most frequent object's relative frequency. It is clear that this object needs at least one occurrence in the sample for the algorithm to be able to find it, and that the sample size must thus scale with $k^s H_{n,s}$.)

7.4 Refinements

When implementing such an algorithm, there are a number of considerations to be taken into account to achieve optimal performance. Perhaps most importantly, one should apply local aggregation when inserting the sample into the distributed hash table to reduce the amount of information that needs to be communicated in practice. We now discuss other potential improvements.

Choice of k^*. In practice, the choice of k^* in Section 7.2 depends on the communication channel's characteristics β, and, to a lesser extent, α, in addition to the problem parameters. Thus, an optimized implementation should take them into account when determining the number of objects to be counted exactly.

Adaptive Two-Pass Sampling (Outline). The objectives of the basic PAC algorithm and its variant using exact counting could be unified as follows: we sample in two passes. In the first pass, we use a small sample size $n\rho_0 = f_0(n, k, \varepsilon, \Delta)$ to determine the nature of the input distribution. From the insights gained from this first sample, we compute a larger sample size $n\rho_1 = f_1(n, k, \hat{c_k}, \varepsilon, \Delta)$. We then determine and return the k most frequent objects of this second sample.

Additionally, we can further refine this algorithm if we can already tell from the first sample that with high probability, there is no slope. If the absolute counts of the objects in the sample are large enough to return the k most frequent objects in the sample with confidence, then taking a second sample would be of little benefit and we can return the k most frequent objects from the first sample.

Using Distributed Bloom Filters. Communication efficiency of the algorithm using exact counting could be improved further by counting sample elements with a distributed single-shot bloom filter (dSBF) [34] instead of a distributed hash table. We transmit their hash values and locally aggregated counts. As multiple keys might be assigned the same hash value, we need to determine the element of rank $k^* + \kappa$ instead of k^*, for some safety margin $\kappa > 0$. We request the keys of all elements with higher rank, and replace the (hash, value) pairs with (key, value) pairs, splitting them where hash collisions occurred. We now determine the element of rank k^* on the $k^* + \kappa + \#\text{collisions}$ elements. If an element whose rank is at most k^* was part of the original $k^* + \kappa$ elements, we are finished. Otherwise, we have to increase κ to determine the missing elements. Observe that if the frequent objects are dominated by hash collisions, this implies that the input distribution is flat and there exist a large number of nearly equally frequent elements. Thus, we may not need to count additional elements in this case.

8 Top-k Sum Aggregation

Generalizing from Section 7, we now consider an input multiset of keys with associated non-negative counts, and ask for the k keys whose counts have the largest sums. Again, the input M is distributed over the p PEs so that no PE has more than $\mathcal{O}(n/p)$ objects. Define $m := \sum_{(k,v)\in M} v$ as the sum of all counts, and let no PE's local sum exceed $\mathcal{O}(m/p)$[1]. We additionally assume that the local input and a key-aggregation thereof—e.g. a hash table mapping keys to their local sums—fit into RAM at every PE.

It is easy to see that except for the sampling process, the algorithms of Section 7 carry over directly, but a different approach is required in the analysis.

8.1 Sampling

Let s be the desired sample size, and define $v_{\text{avg}} := \frac{m}{s}$ as the expected count required to yield a sample. When sampling an object (k, v), its expected sample count is thus $\frac{v}{v_{\text{avg}}}$. To retain constant time per object, we add $\lfloor \frac{v}{v_{\text{avg}}} \rfloor$ samples directly, and one additional sample with probability $\frac{v}{v_{\text{avg}}} - \lfloor \frac{v}{v_{\text{avg}}} \rfloor$ using a Bernoulli trial. We can again use a geometric distribution to reduce the number of calls to the random number generator.

To improve accuracy and speed up exact counting, we aggregate the local input in a hash table, and sample the aggregate counts. This allows us to analyze the algorithms' error independent of the input objects' distribution. A direct consequence is that for each key and PE, the number of samples deviates from its expected value by at most 1, and the overall deviation per key $|s_i - \mathbf{E}[s_i]|$ is at most p.

8.2 Probably Approximately Correct Algorithms

Theorem 15 *We can compute an (ε, δ)-approximation of the top-k highest summing items in expected time $\mathcal{O}\left(\frac{n}{p} + \beta\frac{\log p}{\varepsilon}\sqrt{\frac{1}{p}\log\frac{n}{\delta}} + \alpha\log n\right)$.*

Sampling is done using local aggregation as described in Section 8.1. From then on, we proceed exactly as in Algorithm *PAC* from Section 7.1.

Proof. The part of an element's sample count that is actually determined by sampling is the sum of up to p Bernoulli trials $X_1, \ldots, X_p$ with differing success probabilities. Therefore, its expected value μ is the sum of the probabilities, and we can use Hoeffding's inequality to bound the probability of significant deviations from this value. Let $X := \sum_{i=1}^{p} X_i$. Then,

$$\mathbf{P}[|X - \mu| \geq t] \leq 2e^{-\frac{2t^2}{p}} \quad . \tag{4}$$

We now use this to bound the likelihood that an object has been very mis-sampled. Consider an element j with exact sum $x(j)$ and sample sum s_j. For some threshold Δ, consider the element

[1]This assumption is not strictly necessary, and the algorithm would still be communication efficient without it. However, making this assumption allows us to limit the number of samples transmitted by each PE as s/p for global sample size s. To violate this criterion, a small number of PEs would have to hold $\Omega(s/p)$ elements that are likely to yield at least one sample, making up for a large part of the global sample size without contributing to the result. In such a rare setting, we would incur up to a factor of p in communication volume and obtain running time $\mathcal{O}\left(\frac{n}{p} + \beta\frac{\log p}{\varepsilon}\sqrt{p\log\frac{n}{\delta}} + \alpha\log n\right)$.

mis-sampled if $|x(j) - s_j v_{\text{avg}}| \geq \frac{\Delta}{2}$, i.e. its estimated sum deviates from the true value by more than $\frac{\Delta}{2}$ in either direction. Thus, we substitute $t = \frac{\Delta}{2v_{\text{avg}}}$ into Equation (4) and bound the result by $\frac{\delta}{n}$ to account for all elements. Solving for $s = \frac{m}{v_{\text{avg}}}$, we obtain $s \geq \frac{1}{\varepsilon}\sqrt{2p \ln \frac{2n}{\delta}}$.

In total, we require time $\mathcal{O}\left(\frac{n}{p}\right)$ for sampling, $\mathcal{O}(\beta \frac{s}{p} \log p + \alpha \log p)$ for insertion (as no PE's local sum exceeds $\mathcal{O}(m/p)$, none can yield more than $\mathcal{O}(s/p)$ samples), and $\mathcal{O}(\beta \frac{s}{p} + \alpha \log n)$ for selection. We then obtain the claimed bound as sum of the components. □

As in Section 7, we can use exact summation to obtain a more precise answer. We do not go into details here, as the procedure is nearly identical. The main difference is that a lookup in the local aggregation result now suffices to obtain exact local sums without requiring consultation of the input.

9 Data Redistribution

Let n_i denote the number of data objects present at PE i. Let $n = \sum_i n_i$. We want to redistribute the data such that afterwards each PE has at most $\bar{n} = \lceil n/p \rceil$ objects and such that PEs with more than $\bar{n}$ objects only send data (at most $n_i - \bar{n}$ objects) and PEs with at most $\bar{n}$ objects only receive data (at most $\bar{n} - n_i$ objects). We split the PEs into separately numbered groups of senders s_i and receivers d_i. We also compute the deficit $d_i := \bar{n} - n_i$ on receivers and the surplus $s_i := n_i - \bar{n}$ on senders. Then we compute the prefix sums d and s of these sequences (i.e., $d_i := \sum_{j \leq i} d_j$ and $s_i := \sum_{j \leq i} s_j$). Effectively, d enumerates the empty slots able to receive objects and s enumerates the elements to be moved. Now we match receiving slots and elements to be moved by merging the sequences d and s. This is possible in time $\mathcal{O}(\alpha \log p)$ using Batcher's parallel merging algorithm [7]. A subsequence of the form $\langle d_i, s_j, \ldots, s_{j+k}, d_{i+a}, \ldots, d_{i+b} \rangle$ indicates that sending PEs $j, \ldots, j+k$ move their surplus to receiving PE i (where sending PE $j+k$ only moves its items numbered $s_{j+k-1}+1..d_{i+a}-1$). This is a gather operation. Sending PE $j+k$ moves its remaining elements to receiving PEs $i+a..i+b$. This is a scatter operation. These segments of PE numbers can be determined using segmented prefix operations [8]. Overall, this can be implemented to run in time $\mathcal{O}(\beta \max_i n_i + \alpha \log p)$. Even though this operation cannot remove worst case bottlenecks, it can significantly reduce network traffic.

10 Experiments

We now present an experimental evaluation of the unsorted selection from Section 4.1 and the top-k most frequent objects algorithms from Section 7.

Experimental Setup. All algorithms were implemented in C++11 using OpenMPI 1.8 and Boost.MPI 1.59 for inter-process communication. Additionally, Intel's Math Kernel Library in version 11.2 was used for random number generation. All code was compiled with the `clang++` compiler in version 3.7 using optimization level `-Ofast` and instruction set specification `-march=sandybridge`. The experiments were conducted on InstitutsCluster II at Karlsruhe Institute of Technology, a distributed system consisting of 480 computation nodes, of which 128 were available to us. Each node is equipped with two Intel Xeon E5-2670 processors for a total of 16 cores with a nominal

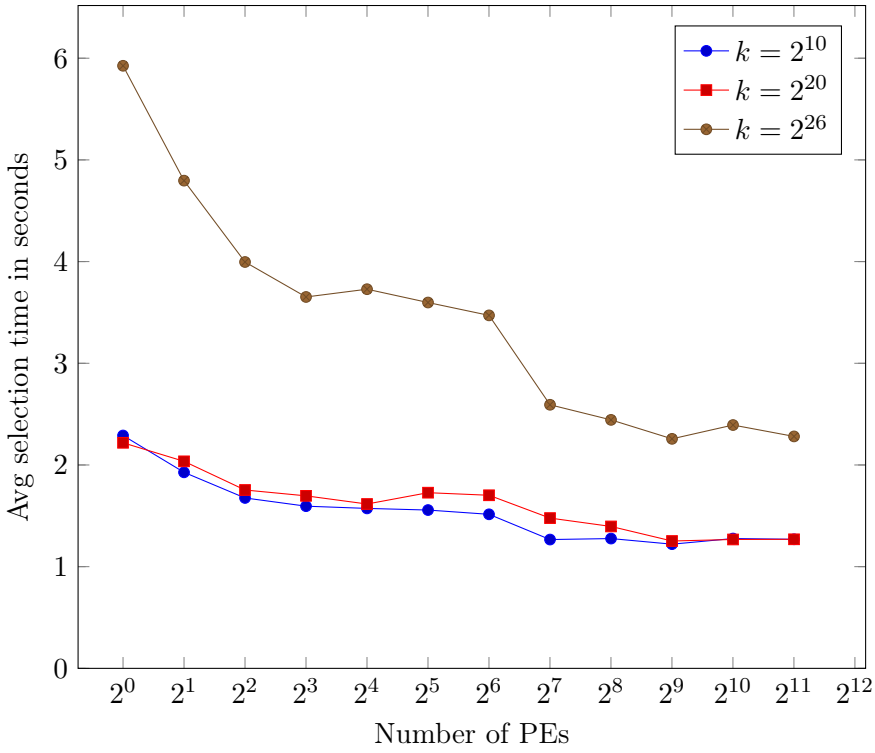


Figure 6: Weak scaling plot for selecting the k-th largest object, $n/p = 2^{28}$, Zipf distribution.

frequency of 2.6 GHz, and 64 GiB of main memory. In total, 2048 cores were available to us. An Infiniband 4X QDR interconnect provides networking between the nodes.

Methodology. We run *weak scaling* benchmarks, which show how wall-time develops for fixed per-PE problem size n/p as p increases. We consider $p = 1$ to 2048 PEs, doubling p in each step. Each PE is mapped to one physical core in the cluster.

Zipf's Law states that the frequency of an object from a multiset M with $|M| = n$ is inversely proportional to its rank among the objects ordered by frequency. Here, we consider the general case with exponent parameter s, i.e. $x_i = n\frac{i^{-s}}{H_{n,s}}$, where $H_{n,s} = \sum_{i=1}^{n} i^{-s}$ is the n-th *generalized harmonic number*.

10.1 Unsorted Selection

Input Generation. We select values from the high tail of Zipf distributions. Per PE, we consider 2^{24}, 2^{26}, and 2^{28} integer elements. To test with non-uniformly distributed data, the PE's distribution parameters are randomized. The Zipf distributions comprise between $2^{20} - 2^{16}$ and 2^{20} elements, with each PE's value chosen uniformly at random. Similarly, the exponent s is uniformly distributed between 1 and 1.2. This ensures that several PEs contribute to the result, so that the distribution is asymmetric, without the computation becoming a local operation at one PE, which has all of the largest elements.

We used several values of k, namely 1024, 2^{20}, and 2^{26}. We do not consider smaller values than 1024, as for values this small, it would be more efficient to locally select the k largest (or smallest) elements, and run a simple distributed selection on those.

Results. Figure 6 shows the results for selecting the k-th largest values from the input, for the above values of k. We can see that in most cases, the algorithm scales even better than the bounds lead us to expect—running time decreases as more PEs are added. This is especially prominent when selecting an element of high rank ($k = 2^{26}$ in Figure 6). The majority of the time is spent in partitioning, i.e. local work, dominating the time spent on communication. This underlines the effect of communication efficiency.

10.2 Top-k Most Frequent Objects

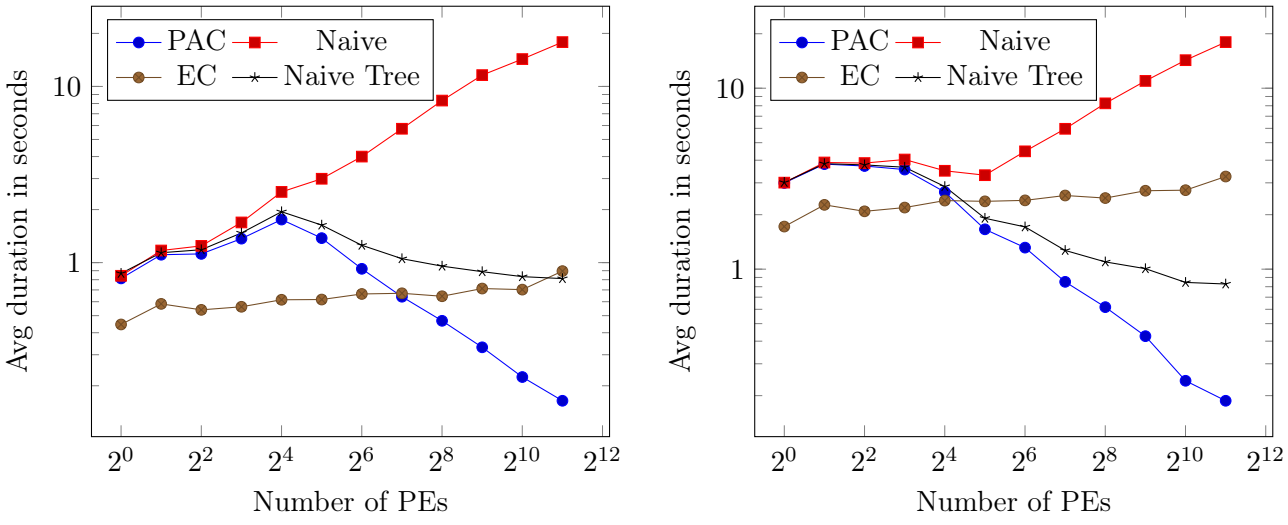
As we could not find any competitors to compare our methods against, we use two naive centralized algorithm as baseline. The first algorithm, *Naive*, samples the input with the same probability as algorithm PAC, but instead of using a distributed hash table and distributed selection, each PE sends its aggregated local sample to a coordinator. The coordinator then uses quickselect to determine the elements of rank $1..k$ in the global sample, which it returns. Algorithm *Naive Tree* proceeds similarly, but uses a tree reduction to send the elements to the coordinator to reduce latency. Similar to Algorithm *PAC*'s hash table insertion operation, this reduction aggregates the counts in each step to keep communication volume low.

Input Generation. We consider 2^{24}, 2^{26} and 2^{28} elements per PE, which are generated according to different random distributions. First, we consider elements distributed according to Zipf's Law with 2^{20} possible values. These values are very concentrated and model word frequencies in natural languages, city population sizes, and many other rankings well [1, 41], the most frequent element being j-times more frequent than that of rank j. Next, we study a negative binomial distribution with $r = 1000$ and success probability $p = 0.05$. This distribution has a rather wide plateau, resulting in the most frequent objects and their surrounding elements all being of very similar frequency. For simplicity, each PE generates objects according to the same distribution, as the distributed hash table into which the sample is inserted distributes elements randomly. Thus, tests with non-uniformly distributed data would not add substantially to the evaluation.

Approximation Quality. To evaluate different accuracy levels, we consider the (ε, δ) pairs $(3 \cdot 10^{-4}, 10^{-4})$ and $(10^{-6}, 10^{-8})$. This allows us to evaluate how running time develops under different accuracy requirements.

We then select the $k = 32$ most frequent elements from the input according to the above requirements. We do not vary the parameter k here, as it has very little impact on overall performance. Instead, we refer to Section 10.1 for experiments on unsorted selection, which is the only subroutine affected by increasing k and shows no increase up to $k = 2^{20}$.

Results. Figure 7 shows the results for 2^{28} elements per PE using $\varepsilon = 3 \cdot 10^{-4}$ and $\delta = 10^{-4}$. We can clearly see that Algorithm *Naive* does not scale beyond a single node at all ($p > 16$). In fact, its running time is directly proportional to p, which is consistent with the coordinator receiving $p - 1$ messages—every other PE sends its key-aggregated sample to the coordinator. Algorithm *Naive*



(a) $n/p = 2^{26}$ (b) $n/p = 2^{28}$

Figure 7: Weak scaling plot for computing the 32 most frequent objects, $\varepsilon = 3 \cdot 10^{-4}$, $\delta = 10^{-4}$. EC suffers from constant overhead for exact counting.

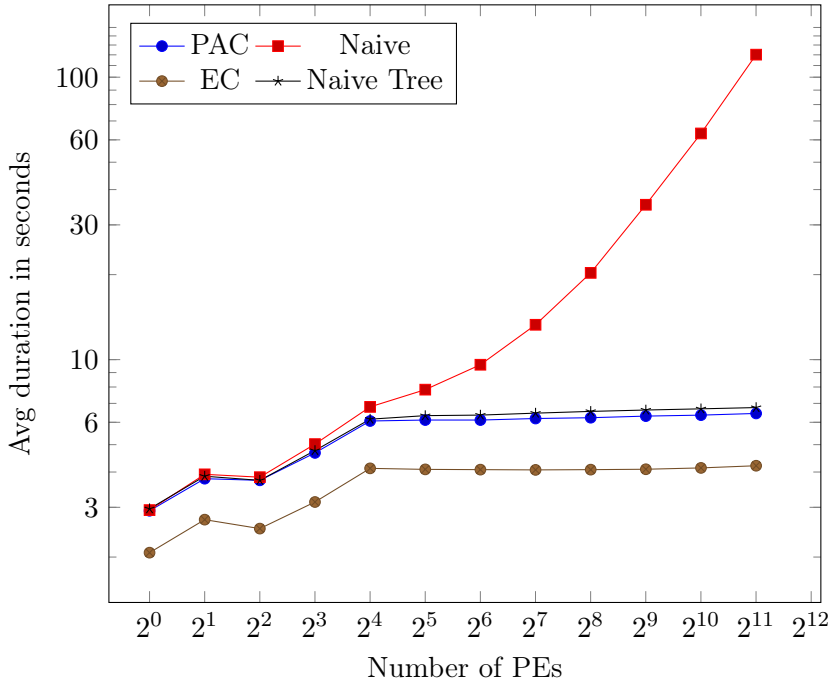


Figure 8: Weak scaling plot for computing the 32 most frequent objects, $n/p = 2^{28}$, $\varepsilon = 10^{-6}$, $\delta = 10^{-8}$. Only EC can use sampling, other methods must consider all objects.

Tree fares better, and actually improves as more PEs are added. This is easily explained by the reduced sample size per PE as p increases, decreasing sampling time. However, communication time begins to dominate, as the decrease in overall running time is nowhere near as strong as the decrease in local sample size. This becomes clear when comparing it to Algorithm *PAC*, which outperforms *Naive Tree* for any number of PEs. We can see that it scales nearly perfectly—doubling the number of PEs (and thereby total input size) roughly halves running time. Since these three algorithms all use the same sampling rate, any differences in running time are completely due to time spent on communication.

Lastly, let us consider Algorithm *EC*. In the beginning, it benefits from its much smaller sample size (see Section 7.2), but as p grows, the local work for exact counting dominates overall running time strongly and Algorithm EC is no longer competitive. Since local work remains constant with increasing p, we see nearly no change in overall running time. To see the benefits of Algorithm *EC*, we need to consider stricter accuracy requirements.

In Figure 8, we consider $\varepsilon = 10^{-6}$ and $\delta = 10^{-8}$. For Algorithms *PAC*, *Naive*, and *Naive Tree*, this requires considering the entire input for any number of PEs, as sample size is proportional to $\frac{1}{\varepsilon^2}$, which the other terms cannot offset here. Conversely, Algorithm *EC*'s sample size depends only linearly on ε, resulting in sample sizes orders of magnitude below those of the other algorithms.

Again, we can see that Algorithm *Naive* is completely unscalable. Algorithm *Naive Tree* performs much better, with running times remaining roughly constant at around 6.5 seconds as soon as multiple nodes are used. Algorithm *PAC* suffers a similar fate, however it is slightly faster at 6.2 seconds. This difference stems from reduced communication volume. However, both are dominated by the time spent on aggregating the input. Lastly, Algorithm *EC* is consistently fastest, requiring 4.1 seconds, of which 3.7 seconds are spent on exact counting[2]. This clearly demonstrates that Algorithm *EC* is superior for small ε.

Smaller local input sizes do not yield significant differences, and preliminary experiments with elements distributed according to a negative binomial distribution proved unspectacular and of little informational value, as the aggregated samples have much fewer elements than in a Zipfian distribution—an easy case for selection.

11 Conclusions

We have demonstrated that a variety of top-k selection problems can be solved in a communication efficient way, with respect to both communication volume and latencies. The basic methods are simple and versatile—the owner-computes rule, collective communication, and sampling. Considering the significant previous work on some of these problems, it is a bit surprising that such simple algorithms give improved results for such fundamental problems. However, it seems that the combination of communication efficiency and parallel scalability has been neglected for many problems. Our methods might have particular impact on applications where previous work has concentrated on methods with a pure master–worker scheme.

It is therefore likely that our approach can also be applied to further important problems. For example, distributed streaming algorithms that generalize the centralized model of Yi and Zhang [40] seem very promising. The same holds for lower bounds, which so far have also neglected multiparty communication with point-to-point communication (see also [34]).

[2]When not all cores are used, memory bandwidth per core is higher. This allows faster exact counting for $p = 1$ to 8 cores on a single node.

Closer to the problems considered here, there is also a number of interesting open questions. For the sorted selection problem from Section 4.2, it would be interesting to see whether there is a scalable parallel algorithm that makes an information theoretically optimal number of comparisons as in the sequential algorithm of Varman et al. [38]. Our analysis of approximate multiselection ignores the case where $\overline{k} - \underline{k} = o(\overline{k})$. It can probably be shown to run in expected time $\mathcal{O}(\alpha \log p \log \frac{\overline{k}}{\overline{k}-\underline{k}})$. For the multicriteria top-k problem from Section 6, we could consider parallelization of advanced algorithms that scan less elements and perform less random accesses, such as [6].

Regarding the top-k most frequent objects and sum aggregation, we expect to be able to conduct fully distributed monitoring queries without a substantial increase in communication volume over our one-shot algorithm.

References

[1] Felix Auerbach. *Das Gesetz der Bevölkerungskonzentration*. 1913.

[2] Michael Axtmann, Timo Bingmann, Peter Sanders, and Christian Schulz. Practical massively parallel sorting. In *27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '15)*, pages 13–23, 2015.

[3] Brian Babcock and Chris Olston. Distributed top-k monitoring. In *ACM SIGMOD 2003*, pages 28–39, 2003.

[4] Brian Babcock and Chris Olston. Distributed top-k monitoring. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 28–39, New York, NY, USA, 2003. ACM.

[5] V. Bala, J. Bruck, R. Cypher, P. Elustondo, A. Ho, C Ho, S. Kipnis, and M. Snir. CCL: A portable and tunable collective communication library for scalable parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):154–164, 1995.

[6] Holger Bast, Debapriyo Majumdar, Ralf Schenkel, Martin Theobald, and Gerhard Weikum. IO-Top-k: Index-access optimized top-k query processing. In *Proceedings of the 32nd international conference on Very large data bases*, pages 475–486. VLDB Endowment, 2006.

[7] Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, pages 307–314, 1968.

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

[9] Shekhar Borkar. Exascale computing – a fact or a fiction? Keynote presentation at IEEE International Parallel & Distributed Processing Symposium 2013, Boston, May 2013.

[10] Pei Cao and Zhe Wang. Efficient top-k query calculation in distributed networks. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 206–215. ACM, 2004.

[11] Michael Christ, James Demmel, Nicholas Knight, Thomas Scanlon, and Katherine Yelick. Communication lower bounds and optimal algorithms for programs that reference arrays – part 1. Technical Report EECS TR No. UCB/EECS-2013-61, UC Berkeley, May 2013.

[12] John Cieslewicz and K.A. Ross. Adaptive Aggregation on Chip Multiprocessors. In *Proc. VLDB Endow.*, pages 339–350, 2007.

[13] N. Deo and S. Prasad. Parallel heap: An optimal parallel priority queue. *Journal of Supercomputing*, 6(1):87–98, 1992.

[14] Ralf Diekmann, Andreas Frommer, and Burkhard Monien. Efficient schemes for nearest neighbor load balancing. *Parallel Computing*, 25(7):789–812, 1999.

[15] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4):614–656, 2003.

[16] Robert W. Floyd and Ronald L. Rivest. Expected time bounds for selection. *Communications of the ACM*, 18(3):165–172, 1975.

[17] Michael T Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In *Algorithms and Computation*, pages 374–383. Springer, 2011.

[18] C. A. R. Hoare. Algorithm 65 (find). *Communication of the ACM*, 4(7):321–322, 1961.

[19] Zengfeng Huang, Ke Yi, and Qin Zhang. Randomized algorithms for tracking distributed count, frequencies, and ranks. In *31st Symposium on Principles of Database Systems (PODS)*, pages 295–306. ACM, 2012.

[20] R. M. Karp and Y. Zhang. Parallel algorithms for backtrack search and branch-and-bound. *Journal of the ACM*, 40(3):765–789, 1993.

[21] T. Leighton. *Introduction to Parallel Algorithms and Architectures*. Morgan Kaufmann, 1992.

[22] Yinan Li, Ippokratis Pandis, Rene Mueller, Vijayshankar Raman, and Guy Lohman. NUMA-aware algorithms: the case of data shuffling. In *CIDR*, 2013.

[23] K. Mehlhorn and P. Sanders. *Algorithms and Data Structures — The Basic Toolbox*. Springer, 2008.

[24] Friedhelm Meyer auf der Heide, Brigitte Oesterdiekhoff, and Rolf Wanka. Strongly adaptive token distribution. *Algorithmica*, 15(5):413–427, 1996.

[25] Sebastian Michel, Peter Triantafillou, and Gerhard Weikum. KLEE: A framework for distributed top-k query algorithms. In *31st International Conference on Very Large Data Bases*, pages 637–648. VLDB Endowment, 2005.

[26] Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.

[27] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. Cache-Efficient Aggregation: Hashing Is Sorting. In *ACM SIMGOD'15*, pages 1123–1136. ACM Press, 2015.

[28] C. G. Plaxton. On the network complexity of selection. In *Foundations of Computer Science*, pages 396–401. IEEE, 1989.

[29] W. H. Press, S.A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 1992.

[30] M. Raab and A. Steger. "balls into bins" – A simple and tight analysis. In *RANDOM: International Workshop on Randomization and Approximation Techniques in Computer Science*, volume 1518, pages 159–170. LNCS, 1998.

[31] P. Sanders. Randomized priority queues for fast parallel access. *Journal Parallel and Distributed Computing, Special Issue on Parallel and Distributed Data Structures*, 1998. extended version: http://www.mpi-sb.mpg.de/~sanders/papers/1997-7.ps.gz.

[32] P. Sanders. Course on Parallel Algorithms, lecture slides, 126–128, 2008. http://algo2.iti.kit.edu/sanders/courses/paralg08/.

[33] P. Sanders, J. Speck, and J. Larsson Träff. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Computing*, 35(12):581–594, 2009.

[34] Peter Sanders, Sebastian Schlag, and Ingo Müller. Communication efficient algorithms for fundamental big data problems. In *IEEE International Conference on Big Data*, 2013.

[35] J. Singler, P. Sanders, and F. Putze. MCSTL: The multi-core standard template library. In *13th Euro-Par*, volume 4641 of *LNCS*, pages 682–694. Springer, 2007.

[36] Amin Vahdat. A look inside Google's data center network. Keynote presentation at Open Networking Summit 2015, Santa Clara.

[37] L. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33(8):103–111, 1994.

[38] Peter J. Varman, Scott D. Scheufler, Balakrishna R. Iyer, and Gary R. Ricard. Merging multiple lists on hierarchical-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):171–177, 1991.

[39] Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas. The lock-free k-LSM relaxed priority queue. *CoRR*, abs/1503.05698, 2015.

[40] Ke Yi and Qin Zhang. Optimal tracking of distributed heavy hitters and quantiles. *Algorithmica*, 65(1):206–223, 2013.

[41] George Kingsley Zipf. *The psycho-biology of language.* The MIT Press, 1965. Originally printed by Houghton Mifflin Co., 1935.

```
(* select object with global rank k *)
Procedure msSelect(s, k)
    if ∑_{1≤i≤p} |s@i| = 1 then                                   -- base case
        return the only nonempty object
    select a pivot v uniformly at random
    find j such that s[1..j] < v and s[j + 1..] ≥ v
if ∑_{1≤i≤p} |j@i| ≥ k then
    return msSelect(s[1..j], k)
else
    return msSelect(s[j_1 + 1..], k − ∑_{0≤i<p} |j@i|)
```

Algorithm 9: Multisequence selection.

A Multisequence Selection [2]

Figure 9 gives high level pseudo code. The base case occurs if there is only a single object (and $k = 1$). We can also restrict the search to the first k objects of each local sequence. A random object is selected as a pivot. This can be done in parallel by choosing the same random number between 1 and $\sum_i |s@i|$ on all PEs. Using a prefix sum over the sizes of the sequences, this object can be located easily in time $\mathcal{O}(\alpha \log p)$. Where ordinary quickselect has to partition the input doing linear work, we can exploit the sortedness of the sequences to obtain the same information in time $\mathcal{O}(\log \sigma)$ with $\sigma := \max_i |s@i|$ by doing binary search in parallel on each PE. If items are evenly distributed, we have $\sigma = \Theta(\min(k, \frac{n}{p}))$, and thus only time $\mathcal{O}(\log \min(k, \frac{n}{p}))$ for the search, which partitions all the sequences into two parts. Deciding whether we have to continue searching in the left or the right parts needs a global reduction operations taking time $\mathcal{O}(\alpha \log p)$. As in ordinary quickselect, the expected depth of the recursion is $\mathcal{O}(\log \sum_i |d_i|) = \mathcal{O}(\log \min(kp, n))$. We obtain the following result.

Theorem 16 *Algorithm 9 can be implemented to run in expected time*

$$\mathcal{O}\left(\left(\alpha \log p + \log \min\left(\frac{n}{p}, k\right)\right) \cdot \log \min\left(kp, n\right)\right) = \mathcal{O}\big(\alpha \log^2 kp\big) \quad .$$

B Running Times of Existing Algorithms

We now prove the running times given for previous works in Table 1.

B.1 Unsorted Selection

Previous algorithms, see [31], rely on randomly distributed input data or they explicitly redistribute the data. Thus, they have to move $\Omega(\frac{n}{p})$ elements in the worst case. The remainder of the bound follows trivially. □

B.2 Sorted Selection

We could not find any prior results on distributed multiselection from sorted lists and thus list a sequential result by Varman et al. [38].

B.3 Bulk Parallel Priority Queue

The result in [31] relies on randomly distributed input data. Therefore, in operation insert*, each PE needs to send its $\mathcal{O}(k/p)$ elements to random PEs, costing $\mathcal{O}((\alpha+\beta)\frac{k}{p})$ time. Then, operation deleteMin* is fairly straight-forward and mostly amounts to a selection. The deterministic parallel heap [13] needs to sort inserted elements and then they travel through $\log \frac{n}{p}$ levels of the data structure, which is allocated to different PEs. This alone means communication cost $\Omega\left(\beta\frac{k}{p}\log p\right)$.

B.4 Heavy Hitters Monitoring

Huang et al. give a randomized heavy hitters monitoring algorithm [19] that requires time $\mathcal{O}\left(\frac{n}{p}+\alpha\frac{\sqrt{p}}{\varepsilon}\log n\cdot\log\frac{\log n}{\delta\varepsilon}\right)$ in our model.

Proof. All communication is between the controller node and a monitor node, thus the maximum amount of communication is at the controller node. Each update, which consists of a constant number of words, is transmitted separately. Thus, the communication term given by the authors transfers directly into our model (except that the number of monitor nodes k is $p-1$ here). Making the algorithm correct for all time instances and increasing the success probability to arbitrary values of δ accounts for the $\log\frac{\log n}{\delta\varepsilon}$ factor [19]. □

B.5 Top-k Frequent Objects Monitoring

Monitoring Query 1 of [3] performs top-k most frequent object monitoring, for which it incurs a running time of $\Omega\left(\frac{n}{p}+\beta\frac{k}{\varepsilon}+\alpha\frac{1}{\varepsilon}\right)$ for relative error bound $\varepsilon=\frac{n}{\Delta}$, where Δ corresponds to ε in their notation. This algorithm also has further restrictions: It does not provide approximate counts of the objects in the top-k set. It can only handle a small number N of distinct objects, all of which must be known in advance. It requires $\Theta(Np)$ memory on the coordinator node, which is prohibitive if N and p are large. It must also be initialized with a top-k set for the beginning of the streams, using an algorithm such as TA [15]. We now present a family of inputs for which the algorithm uses the claimed amount of communication.

Initialize with $N=k+2$ items $O_1,\ldots,O_{k+2}$ that all have the same frequency. Thus, the initial top-k set comprises an arbitrary k of these. Choose one of the two objects that are *not* in the top-k set, and refer to this object as O_s, and pick a peer (PE) N_f. Now, we send O_s to N_f repeatedly, and all other items to all other peers in an evenly distributed manner (each peer receives around the same number of occurrences of each object). After at most 2Δ steps, the top-k set has become invalid and an expensive full resolution step is required[3]. As we expect Δ to be on the large side, we choose $F_0=0$ and $F_j=\frac{1}{p-1}$ for $j>0$ as per the instructions in [3]. We can repeat this cycle to obtain an instance of the example family for any input size n. Note that the number of "cheap" resolution steps during this cycle depends on the choice of the F_j values, for which Babcock and

[3]This actually happens much earlier, the first "expensive" resolution is required after only $\mathcal{O}(\frac{\Delta}{p})$ steps. This number increases, but will never exceed $\sum_{i=0}^{\infty}\Delta(k+1)^{-i}\leq 2\Delta$.

Olston give rough guidelines of what might constitute a good choice, but do not present a theoretical analysis of how it influences communication. Here, we ignore their cost and focus solely on the cost of "expensive" resolutions.

By the above, an "expensive" resolution round is required every (at most) $\mathcal{O}(p\Delta)$ items in the input. Since the resolution set contains $k+1$ objects (the top-k set plus the non-top-k object with a constraint violation), each "expensive" resolution has communication cost $\Theta(\beta kp + \alpha p)$. Thus, we obtain a total communication cost for expensive resolutions of $\Omega(\frac{1}{p\varepsilon}(\beta kp + \alpha p)) = \Omega(\beta\frac{k}{\varepsilon} + \alpha\frac{1}{\varepsilon})$, for a given relative error bound δ. Additionally, each item requires at least constant time in local processing, accounting for the additive $\Omega(\frac{n}{p})$ term. The actual worst-case communication cost is likely even higher, but this example suffices to show that the approach of [3] is not communication-efficient in our model. □

B.6 Multicriteria Top-k

Previous algorithms such as TPUT [10] or KLEE [25] are not directly comparable to our approach for a number of reasons. First, they do not support arbitrary numbers of processing elements, but limit p to the number of criteria m. Each PE is assigned one or more complete index lists, whereas our approach splits the *objects* among the PEs, storing the index lists of the locally present objects on each PE. This severely limits TPUT's and KLEE's scalability. Secondly, as noted in the introduction, these algorithms use a centralized master–worker approach, where the master (or *coordinator*) node handles *all* communication. This further limits scalability and leads to an inherent increase in communication volume by a factor of up to p. Thirdly, they are explicitly designed for *wide-area networks* (WANs), whereas our algorithms are designed with strongly interconnected PEs in mind, as they might be found in a data center. Since the modeling assumptions are too different to provide a meaningful comparison, we refrain from giving a communication analysis of these algorithms (nor was one provided in the original papers).