Algorithmica manuscript No.
(will be inserted by the editor)

A Parallel Space Saving Algorithm For Frequent Items and the Riemann–Hurwitz zeta distribution

Massimo Cafaro · Piergiulio Tempesta · Marco Pulimeno

Received: date / Accepted: date

Abstract A parallel version of the Space Saving algorithm to solve the k–majority problem is presented. The algorithm determines in parallel frequent items, i.e., those whose multiplicity is greater than a given threshold, and is therefore useful for iceberg queries and many other different contexts. We apply our algorithm to the detection of frequent items in a stream of data whose probability distribution function is a Riemann–Hurwitz or Zipf–Mandelbrot distribution.

Keywords frequent items · space saving algorithm · message–passing

1 Introduction

The purpose of this paper is to offer a solution to the problem of determining *frequent items* in a stream of data, based on a Space Saving parallel algorithm. Given a stream of data $\mathcal{A}$ of n elements, and an integer $2 \le k \le n$, this problem requires finding the set of elements whose multiplicity is greater than $\left\lfloor \frac{n}{k} \right\rfloor$. It can be stated formally as follows. We begin by recalling some basic definitions of multiset theory [47].

M. Cafaro
University of Salento, Lecce, Italy
CMCC – Euro–Mediterranean Center for Climate Change, Lecce, Italy
E-mail: massimo.cafaro@unisalento.it

P. Tempesta
Departamento de Física Teórica II (Métodos Matemáticos de la física), Facultad de Físicas, Universidad Complutense de Madrid, 28040 – Madrid, Spain
E-mail: p.tempesta@fis.ucm.es

M. Pulimeno
University of Salento, Lecce, Italy
E-mail: mpulimeno@gmail.com

Definition 1 A multiset $\mathcal{A} = (A, m_{\mathcal{A}})$ is a pair where A is some set, called the underlying set of elements, and $m_{\mathcal{A}} : A \to \mathbb{N}$ is a function. The generalized indicator function of $\mathcal{A}$ is

$$I_{\mathcal{A}}(x) := \begin{cases} m_{\mathcal{A}}(x) & x \in A, \\ 0 & x \notin A, \end{cases} \tag{1}$$

where the integer–valued function $m_{\mathcal{A}}$, for each $x \in A$, provides the *multiplicity* or number of occurrences of x in $\mathcal{A}$. The cardinality of $\mathcal{A}$ is expressed by

$$Card(\mathcal{A}) = \sum_{x \in A} I_{\mathcal{A}}(x) := |\mathcal{A}|. \tag{2}$$

A multiset (also called a *bag*) essentially is a set where the duplication of elements is allowed. In the sequel, $\mathcal{A}$ will play the role of a finite input array, containing n elements. We can now state the problem formally.

Definition 2 Given a *multiset* $\mathcal{A}$, with $|\mathcal{A}| = n$, a k–majority element (or *frequent item*) is an element $x \in \mathcal{A}$ whose *multiplicity* $m_{\mathcal{A}}(x)$ is such that $m_{\mathcal{A}}(x) \geq \left\lfloor \frac{n}{k} \right\rfloor + 1$.

Statement of the Problem. *The k–majority problem takes as input an array $\mathcal{A}$ of n numbers, and requires as output the set $W = \left\{x : m_{\mathcal{A}}(x) \geq \left\lfloor \frac{n}{k} \right\rfloor + 1\right\}$.*

Therefore, the k–majority problem entails finding the set of elements whose multiplicity is greater than a given threshold controlled by the parameter k. It is worth noting here that when $k = 2$, the problem reduces to the well known majority problem [9], [10], [25].

Determining frequent items is a problem that appears in the literature in many different contexts. In data mining the problem is usually associated to two contexts, the on–line (stream) and the off–line setting, the difference being that in the former case we are restricted to a single scan of the input. In practice, this implies that verifying the frequent items that have been found in order to discard false positives is not allowed, while in the latter case a parallel scan of the input can be used to assess the actual k–majority elements. Finding frequent items is also referred to as *hot list analysis* [27] or market basket analysis [12].

In the context of data bases, the problem is usually called an *iceberg query* [24], [6]. The name arises from the fact that the number of k–majority elements is often very small (the tip of an iceberg) when compared to the large amount of input data (the iceberg).

Other practical uses of frequent items include, for instance, electronic voting, where a quorum of more than n/k of all the votes in a ballot is required for a candidate to win; additional examples deals with network traffic analysis, in which the extraction of essential features of traffic streams passing through internet routers requires frequency estimation of internet packets [21]. Another example is monitoring internet packets in order to infer network congestion [23], [45]. In [33], tracking the identities of the most frequently accessed items in networks distributing contents is done through a Gossip based algorithm. Determining frequent items in P2P networks is discussed in [34], using an approach called in–network filtering.

The problem also arise in the context of the analysis of web query logs [14], and is relevant in Computational Linguistics, for instance in connection with the estimation of the frequencies of specific words in a given language [1], or in all contexts where a verification of the Zipf–Mandelbrot law is required [52], [35] (theoretical linguistics, ecological field studies [41], etc.). We note here that the class of applications considered here is characterized by the condition $k = O(1)$.

The first sequential algorithm solving the k–majority problem was designed by Misra and Gries [40]. Their solution, whose worst-case complexity is $O(n \log k)$, is based on the following idea. Given a multiset, repeatedly deleting k distinct elements from it until possible, leads to a k–reduced multiset containing exactly all the elements of the initial multiset which are k–majority candidates. The algorithm's complexity is strictly related to the particular data structure used, which is an AVL tree [2].

About twenty years later, Demaine et al. [21] and Karp et al. [31] rediscovered independently the Misra and Gries algorithm, and published almost simultaneously optimal algorithms. *Frequent*, the algorithm designed by Demaine et al. exploits better data structures (a doubly linked list of groups, supporting decrementing a set of counters at once in $O(1)$ time) and achieves a worst-case complexity of $O(n)$. The algorithm devised by Karp et al. is based on hashing and therefore achieves the $O(n)$ bound on average.

Cormode and Hadjieleftheriou present in [18] a survey of existing algorithms, which are classified as *counter* or *sketch* based. All of the algorithms we have discussed so far belong to the former class; notable examples of counters–based algorithms developed recently include *LossyCounting* [38] and *SpaceSaving* [39]. Among the sketch–based ones, we recall here *CountSketch* [14] and *CountMin* [19]. It is worth noting here the interesting observation made by Cormode and Hadjieleftheriou in the concluding remarks: "In the distributed data case, different parts of the input are seen by different parties (different routers in a network, or different stores making sales). The problem is then to find items which are frequent over the union of all the inputs. Again due to their linearity properties, sketches can easily solve such problems. It is less clear whether one can merge together multiple counter–based summaries to obtain a summary with the same accuracy and worst–case space bounds".

In [51], an algorithm based on an extensible and scalable Bloom Filter is presented. A related problem, determining frequent items over a sliding window is discussed in [29]; the authors present an algorithm with constant update time. In [49], an approach to mining top–k frequent items over sliding windows whose length changes dynamically has been investigated.

In the distributed setting, recent work related to frequent items include [37], [15], [4] and [26]; other work strictly related to distributed monitoring of data streams include [16], [44] and [32].

It is worth noting here that, in the parallel setting, just few works have been published. In [48], FPGAs (Field Programmable Gate Arrays) hardware implementations of frequent items computations are presented. In [20], a shared-memory parallel algorithm for frequent items is designed for multicore architectures using a cooperation–based locking approach among threads. In a recent paper of us [13], we presented a parallel version of the *Frequent* algorithm, showing how to merge in parallel multiple counter–based summaries.

In this paper, we investigate how to parallelize the *SpaceSaving* algorithm, which considerably improves space requirements with regard to Frequent. We design the algorithm in the context of message–passing architectures, prove the correctness of the algorithm, and then analyze its parallel complexity proving its cost–optimality for $k = O(1)$. The main motivation to parallelize the Space Saving algorithm is that, as proved in [18] and [36], this algorithm appears better than other counter-based algorithms, across a wide range of data types and parameters. From a technical perspective, parallelizing this algorithm has been more challenging with regard to the Frequent algorithm, since, as discussed in Section 3, merging summaries in parallel is trickier.

Another original aspect of this work is that we apply our algorithm to the study of frequent items in a large set of data, whose probability distribution function is a Riemann–Hurwitz distribution. This distribution generalizes the classical Zipf distribution and is directly related to important models of networks recently introduced by [5]: the *scale–free complex networks*. These networks are defined on suitable *random graphs*. The attachment probability among different nodes of a network is indeed described by a Riemann–Hurwitz distribution. As shown in Section 5, experimental results confirm exceptional scalability and performances of our parallel algorithm in all of the cases we investigated, so that it is especially suitable for treating these kind of streams of data.

This article is organized as follows. Our parallel space saving algorithm is presented in Section 2. We prove its correctness in Section 3, analyze it and prove its cost–optimality for $k = O(1)$ in Section 4. We conclude by presenting in Section 5 experimental results concerning the application of our algorithm to a stream of data governed by a Zipf–Mandelbrot distribution and by a Hurwitz distribution.

2 A Parallel Space Saving Algorithm

Let p be the number of processors we use in parallel. The pseudocode of Fig. 1 describes our parallel Space Saving algorithm. We assume that the array $\mathcal{A}$ is initially read by an application calling our function implementing the algorithm; for instance, every process reads the input from a file or a designated process reads it and broadcast it to the other processes. The initial call is *ParallelSpaceSaving* $(\mathcal{A}, n, p)$, where $\mathcal{A}$ is an input array consisting of n elements. The algorithm determines k–majority candidates. We recall here that, indeed, some of the candidates returned may be false positives as in the sequential counterpart.

The algorithm works as follows. The initial domain decomposition is done in steps 1–2. Each processor determines the indices of the first and last element related to its block, by applying a simple block distribution, in which each processor is responsible for either $\lfloor n/p \rfloor$ or $\lceil n/p \rceil$ elements.

Then, each processor determines *local*, an array of structures storing its local candidates, their corresponding occurrences and errors in step 3, by utilizing the well–known algorithm designed by Metwally et al. [39], shown in the pseudocode as the *SpaceSaving* function. The occurrences are the values associated to the counters greater than zero corresponding to the k–majority candidates determined, and the errors are the ϵ values as described in the sequential Space Saving algorithm.

The *local* data is used as input for the parallel reduction of step 4, whose purpose is to determine global candidates for the whole array. This step is carried out by means of the $ParallelCandidateReduction$ function, shown in the text as Fig. 2.

Assuming that at the end of the parallel reduction the processor whose rank is one holds the result of the parallel reduction, steps 5–7 carried out by the processor with rank one either return the set of potential k–majority elements or the empty set.

The parallel reduction determines global candidates for the whole array and works as shown in Fig. 2. At each step of the reduction, a processor receives as input from two processors p_i and p_j, their arrays of structures $local_i$ and $local_j$, which hold local candidate items, their occurrences and their errors. The reduction step is very similar to the sequential Space Saving algorithm, with two major differences: we correct the counters' values by subtracting in each array the minimum number of hits associated to the counters in that array, and then update each item by considering one-shot all of its occurrences instead of dealing with them one at a time. The former is required for the correctness of the algorithm, while the latter is for performance reasons. These changes are discusses in detail in Sections 3 and 4.

We begin initializing a stream summary data structure in step 1. Then, we determine in each input array the minimum number of hits associated to the counters in steps 2–3. Owing to the fact that counters are maintained in sorted order in the stream summary data structure, the minimum values are kept in the first counter of the input arrays (which are the output of the sequential Space Saving algorithm run immediately before engaging in the parallel reduction step). For each input array, we update the stream summary data structure. This is done respectively in steps 4–6 and 7–9. Each item is updated using StreamSummaryUpdate, shown in the text as Fig. 3. Here, the update works as in the traditional sequential Space Saving algorithm, except that we also keep track of the ϵ_p error induced by the parallel processing (which is strictly related to the need to subtract the minimum in order to guarantee the algorithm's correctness) and increment a counter using a number of occurrences $count \geq 1$ (for performance reasons).

Fig. 1: Parallel Space Saving Algorithm

```
Require: $\mathcal{A}$, an array; $n$, the length of $\mathcal{A}$; $p$, the number of processors
Ensure: an array containing $k$–majority candidate elements {The $n$ elements of the input
    array $\mathcal{A}$ are distributed to the $p$ processors so that each one is responsible for either
    $\lfloor n/p \rfloor$ or $\lceil n/p \rceil$ elements; let $left$ and $right$ be respectively the indices of the first and
    last element of the sub-array handled by the process with rank $id$}
 1: $left \leftarrow \lfloor (id-1)\ n/p \rfloor$
 2: $right \leftarrow \lfloor id\ n/p \rfloor - 1$
    {determine local candidates}
 3: $local \leftarrow SpaceSaving(\mathcal{A}, left, right)$
    {determine the global candidates for the whole array}
 4: $global \leftarrow ParallelCandidateReduction(local)$
    {we assume here that the processor with rank 1 holds the final result of the parallel
    reduction}
 5: if $id == 1$ then
 6:     return $global$
 7: end if
```

Fig. 2: ParallelCandidateReduction Algorithm

```
Require: $local_i$ and $local_j$, arrays of structures holding local candidate items, their occur-
    rences and their errors for processors $p_i$ and $p_j$
Ensure: arrays containing $k$–majority candidate elements and their weights
 1: $streamSummary \leftarrow StreamSummaryInit()$
 2: $min_i \leftarrow local_i[0].counter$
 3: $min_j \leftarrow local_j[0].counter$
 4: for $z = 0$ to $k-1$ do
 5:     $StreamSummaryUpdate(streamSummary, local_i[z].item,$
        $local_i[z].counter - min_i, local_i[z].\epsilon - min_i, local_i[z].\epsilon_p)$
 6: end for
 7: for $z = 0$ to $k-1$ do
 8:     $StreamSummaryUpdate(streamSummary, local_j[z].item,$
        $local_j[z].counter - min_j, local_j[z].\epsilon - min_j, local_j[z].\epsilon_p)$
 9: end for
10: $output \leftarrow GetResults(streamSummary)$
11: for $z = 0$ to $k-1$ do
12:     $output[z].\epsilon \leftarrow output[z].\epsilon + min_i + min_j$
13:     $output[z].\epsilon_p \leftarrow output[z].\epsilon p + min_i + min_j$
14: end for
15: return $output$
```

At the beginning of the StreamSummaryUpdate, we check the *count* value of the *item* to be used to update the *streamSummary* data structure. If this value is zero, then we simply skip the update and return (steps 1–3). If a counter c in the *streamSummary* data structure is already in charge of monitoring the input element *item*, we simply update the item's error fields ϵ and ϵ_p and increment one-shot the counter's occurrences by adding *count* occurrences (steps 4–7). Otherwise (steps 9–14), once the counter m monitoring the item with least hits *min* has been found, we update the counter so that it starts monitoring the input item, update the error ϵ as in the sequential Space Saving algorithm, update the parallel error ϵ_p and finally increment one-shot the counter's occurrences by adding *count* occurrences.

Fig. 3: StreamSummaryUpdate Algorithm

```
Require: streamSummary, the data structure to be updated; item, the item to be used;
    count, the occurrences of item; $\epsilon$, the error associated to item; $\epsilon_p$, the parallel error
    associated to item;
Ensure: one-shot update of the streamSummary data structure
 1: if $count == 0$ then
 2:     return
 3: end if
 4: if item is monitored by a counter $c$ in streamSummary then
 5:     $c.\epsilon \leftarrow c.\epsilon + \epsilon$
 6:     $c.\epsilon_p \leftarrow c.p\epsilon + \epsilon_p$
 7:     $incrementCounter(c, count)$
 8: else
 9:     let $m$ be a counter in streamSummary monitoring the item with least hits
10:     let $min$ be the number of hits for the counter $m$
11:     $m.item \leftarrow item$
12:     $m.\epsilon \leftarrow min + \epsilon$
13:     $m.\epsilon_p \leftarrow \epsilon_p$
14:     $incrementCounter(m, count)$
15: end if
16: return
```

3 Formal Correctness

In this Section we prove the correctness of our parallel Space Saving algorithm. We decompose the original array (i.e. multiset) of data $\mathcal{A}$ in p subarrays $\mathcal{A}_i$ ($i = 1, \dots, p$), namely $\mathcal{A} = \biguplus_i \mathcal{A}_i$. Here the $\uplus$ operator denotes the *join operation* [47], which is the sum of the multiplicity functions as follows: $I_{\mathcal{A} \uplus \mathcal{B}}(x) = I_{\mathcal{A}}(x) + I_{\mathcal{B}}(x)$. Let the sub–array $\mathcal{A}_i$ be assigned to the processor p_i, whose rank is denoted by id, with $id = 1, \dots, p$. Let also $|\mathcal{A}_i|$ denote the cardinality of $\mathcal{A}_i$, with $\sum_i |\mathcal{A}_i| = |\mathcal{A}| = n$. We begin by proving in the following Lemmas some basic facts about k–majority elements.

Lemma 1 *Given an array $\mathcal{A}$ of n elements and $2 \leq k \leq n$, there are at most $k-1$ distinct k–majority elements.*

Proof By contradiction, assume that there are at least k distinct k–majority elements. It follows that the array $\mathcal{A}$ must contain at least $k\ (\lfloor n/k \rfloor + 1) > k\ (n/k) = n$ elements, thus contradicting the hypothesis that $|\mathcal{A}| = n$.

Lemma 2 *Let $x \in \mathcal{A}$ be an element such that $I_{\mathcal{A}_i}(x) \leq n/(p\ k)$, $\forall i \in \{1, \dots, p\}$. Then, the element x can not be a k–majority element.*

Proof It suffices to observe that

$$\sum_{i=1}^{p} I_{\mathcal{A}_i}(x) \leqslant \frac{n}{k} < \frac{n}{k} + 1. \tag{3}$$

The first step of the algorithm consists in the execution of the sequential Space Saving algorithm, which has already been proved to be correct by its authors, on the subarray assigned to each processor p_i. Therefore, in order to prove the overall correctness of the algorithm, we just need to demonstrate that the parallel reduction is correct. Our strategy is to prove that a single step of the parallel reduction is correct, then to extend the proof to the $O(\log\ p)$ steps of the whole parallel reduction and, finally, to prove that going back from the initial input of the parallel reduction to the algorithm's input A preserves the correctness.

We recall here a few basics facts related to the sequential Space Saving algorithm that will be used later. Let $\mathcal{W}$ be the output of the algorithm. We consider it as a multiset $\mathcal{W} = (W, m)$ in which $W = \{c_1, \dots, c_j\}$, $j \in \{1, \dots, k\}$, and the associated indicator function returns the counters' occurrences.

Observe that the cardinality of W is at most k, since k is the maximum number of counters used during the execution of the algorithm. Instead, the cardinality of the multisets $\mathcal{A}$ and $\mathcal{W}$ coincide:

$$|W| \leq k \quad \text{and} \quad |\mathcal{A}| = |\mathcal{W}| \tag{4}$$

The algorithm correctly reports all the k–majority elements, and, for each one, it provides an upper bound on its multiplicity in $\mathcal{A}$:

$$\forall x \in \mathcal{A} \colon m_{\mathcal{A}}(x) \geq \left\lfloor \frac{|\mathcal{A}|}{k} \right\rfloor + 1 \Rightarrow x \in \mathcal{W} \tag{5}$$

$$\forall x \in \mathcal{W} \colon m_{\mathcal{W}}(x) \geq m_{\mathcal{A}}(x) \tag{6}$$

From eqs. (4), (5), (6) it follows that if an element x is of k–majority for the input $\mathcal{A}$, then it must be necessarily of k–majority for the output $\mathcal{W}$.

The case $|A| \leq k$ is trivial; all the distinct input elements are monitored and reported with their multiplicities by the counters.

When $|A| > k$ the algorithm correctly reports all the elements whose frequencies exceed the minimum among the counters. This minimum is bounded by $\left\lfloor \frac{|\mathcal{A}|}{k} \right\rfloor$:

$$\forall x \in \mathcal{A} \colon m_{\mathcal{A}}(x) > \min_{y \in \mathcal{W}} \{m_{\mathcal{W}}(y)\} \Rightarrow x \in \mathcal{W}, \tag{7}$$

with

$$\min_{y \in \mathcal{W}} \{m_{\mathcal{W}}(y)\} \leq \left\lfloor \frac{|\mathcal{A}|}{k} \right\rfloor. \tag{8}$$

At the j–th step of the parallel reduction, given as an input two multisets $\mathcal{S}_1^j$ and $\mathcal{S}_2^j$, the reduction outputs a multiset $\mathcal{W}_j$. The cardinality of S_1 and S_2 is at most k in each step. Indeed, this is true at the beginning of the reduction, since the input is generated by applying the sequential Space Saving algorithm. Then, we maintain this property in the subsequent steps by computing $\mathcal{W}_j$ as follows.

We start by constructing an intermediate multiset $\mathcal{M}$ from $\mathcal{S}_1$ and $\mathcal{S}_2$, that will serve as input in the j–th reduction step (here we omit the index j for sake of simplicity). The multiset M is constructed by using a *modified join operation*, defined as follows.

Definition 3 Given two multisets, we call modified join operation the composition rule

$$I_{\mathcal{S}_1 \underset{\mathcal{M}}{\uplus} \mathcal{S}_2}(x) = \begin{cases} I_{\mathcal{S}_1}(x) + I_{\mathcal{S}_2}(x) - \mu_{\mathcal{S}_1} - \mu_{\mathcal{S}_2} & x \in \mathcal{S}_1 \cap \mathcal{S}_2 \\ I_{\mathcal{S}_1}(x) - \mu_{\mathcal{S}_1} & x \in \mathcal{S}_1 \setminus \mathcal{S}_2 \\ I_{\mathcal{S}_2}(x) - \mu_{\mathcal{S}_2} & x \in \mathcal{S}_2 \setminus \mathcal{S}_1 \end{cases} \tag{9}$$

where, for $i = 1, 2$:

$$\mu_{S_i} = \begin{cases} \min\limits_{y \in S_i} m_{S_i(y)} & \text{if} \quad |A_i| \geq k, \\ 0 & \text{otherwise.} \end{cases} \tag{10}$$

Therefore we define the underlying set of elements M of the intermediate multiset $\mathcal{M}$ as

$$M = \cup \, \{y \in S_i \mid m_{S_i(y)} > \mu_{S_i}, \quad i = 1, 2\}, \tag{11}$$

with $I_{S_1 \underset{\mathcal{M}}{\uplus} S_2}(x) = I_M = \begin{cases} m_{\mathcal{M}}(x) & x \in \mathcal{M}, \\ 0 & x \notin \mathcal{M}. \end{cases}$
It is worth noting here that the elements that are not reported in $\mathcal{M}$ cannot be k–majority elements due to eq. (8) and Lemma 2.

Remark. The *modification* to the Space Saving algorithm we perform in (9) is a pre–processing step required to preserve the property of an element of being of k–majority. Indeed, this property may not hold anymore when in one or more of the multisets $\mathcal{A}_i$ we evict an element from a counter, replace it with a new element and increment the counter. Instead, if no replacements are made, the property holds. The correctness of (9) rests on the following Lemma, used in the Misra–Gries [40] and Frequent [21] algorithms: *Given a multiset $\mathcal{A}$ of n elements, repeatedly deleting k–uples of distinct elements from it until no longer possible leads to a k–reduced multiset, i.e. a multiset containing only elements equal to the k–majority candidates or the empty set.* In [13] we proved that this strategy works correctly in parallel.

In each reduction step, then we run on each processor our modified version of the Space Saving algorithm. It is essentially equivalent to the sequential algorithm; the only difference is the way we handle the elements in order to achieve a lower complexity, by taking advantage of the input structure.

Let us prove the formal correctness of the overall procedure. Consider first the case $p = 2$, which entails that a single reduction step is performed.

Theorem 1 *If an element $x \in \mathcal{A}$ is a k–majority element for $\mathcal{A}$, then both $x \in \mathcal{W}$ and x is a k–majority element for $\mathcal{W}$.*

Proof By contradiction, assume that either $x \notin \mathcal{W}$ or $x \in \mathcal{W}$ and x is not a k–majority element for $\mathcal{W}$. We let the Space Saving sequential algorithm process the multisets $\mathcal{A}_1$ and $\mathcal{A}_2$ arising from the initial domain decomposition. Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be the resulting multisets.

Let us consider first the case $x \notin \mathcal{W}$. As a consequence of (7)

$$m_{\mathcal{M}}(x) \leq \left\lfloor \frac{|\mathcal{M}|}{k} \right\rfloor . \tag{12}$$

Therefore, by using (9) and (10)

$$m_{\mathcal{M}}(x) \leq \left\lfloor \frac{|\mathcal{S}_1| + |\mathcal{S}_2|)}{k} \right\rfloor - (\mu_{\mathcal{S}_1} + \mu_{\mathcal{S}_2}), \tag{13}$$

where we took into account that either we subtract k times the minimum on the sets S_i or the value zero. Equivalently:

$$m_{\mathcal{M}}(x) + (\mu_{\mathcal{S}_1} + \mu_{\mathcal{S}_2}) \leq \left\lfloor \frac{|\mathcal{A}|}{k} \right\rfloor . \tag{14}$$

Let us introduce the function

$$\bar{m}_{\mathcal{X}}(x) = \begin{cases} m_{\mathcal{X}}(x) & \text{if } x \in \mathcal{X} & (15a) \\ \mu_{\mathcal{X}} & \text{otherwise,} & (15b) \end{cases}$$

where $\mathcal{X}$ is a multiset. From the Definition 3 one can deduce that, for any $x \in S_1$ or S_2 (and consequently in the merge set $\mathcal{M}$), the following relation holds

$$\bar{m}_{\mathcal{S}_1}(x) + \bar{m}_{\mathcal{S}_2}(x) = m_{\mathcal{M}}(x) + \mu_{\mathcal{S}_1} + \mu_{\mathcal{S}_2}. \tag{16}$$

Since by hypothesis x is a k–majority element for $\mathcal{A}$, due to Lemma 2 x is of k–majority in at least one of the two multisets $\mathcal{A}_1$ and $\mathcal{A}_2$, for instance in $\mathcal{A}_1$. Therefore, $m_{\mathcal{A}_1}(x) \leq \bar{m}_{\mathcal{S}_1}(x)$, as a consequence of the monotony property (6). Also,

$$m_{\mathcal{A}_2}(x) \leq \bar{m}_{\mathcal{S}_2}(x), \tag{17}$$

because either $x \in S_2$, and then by monotony the equation (17) holds, or $x \notin S_2$. In this case, due to the properties of the Space Saving algorithm, again $m_{\mathcal{A}_1}(x) \leq \mu_{\mathcal{S}_1}(x)$, which guarantees the validity of eq. (17). Therefore, we deduce:

$$m_{\mathcal{A}}(x) = m_{\mathcal{A}_1}(x) + m_{\mathcal{A}_2}(x) \leq \bar{m}_{\mathcal{S}_1}(x) + \bar{m}_{\mathcal{S}_2}(x) \leq \left\lfloor \frac{|\mathcal{A}|}{k} \right\rfloor , \tag{18}$$

which is against the assumption that x is a k–majority element for $\mathcal{A}$.

The second case, i.e. $x \in \mathcal{W}$ and x is not a k–majority element for $\mathcal{W}$, still leads to eqs. (12) and (13). Consequenty, the proof in this case is completely analogous to the previous one.

Now we extend the proof to a general reduction procedure consisting of $\lceil \log_2 p \rceil$ steps.

Theorem 2 *The proposed algorithm determines correctly all the k–majority candidate elements.*

Proof Assume without loss of generality that $p = 2^r$, with $r \in \mathbb{N}$. Then the parallel reduction consists of r steps, ranging from $j = 0$ to $j = r - 1$. Assume that at least one k–majority element x exists for the input multiset $\mathcal{A}$. In the j–th step of the parallel reduction, we process $p/2^j$ input multisets and provide in output $p/2^{j+1}$ multisets that, in turn, will be the input for the $(j + 1)$–th step. Let $q = p/2^{j+1}$, and $\mathcal{S}_i^{(j)} (i = 1, \ldots, 2q)$ and $\mathcal{W}_s^{(j)} (s = 1, \ldots, q)$ be the input and the output of the $\biguplus_{\mathcal{M}}$ operator in the j–th step, respectively. Let $\mathcal{W}_1^{(r-1)} \equiv \mathcal{W}^{(r-1)}$ be the final multiset outputted by the parallel algorithm and $\mathcal{S}_1^{(r-1)}$, $\mathcal{S}_2^{(r-1)}$ be the input of the final reduction step.

By contradiction, suppose that either $x \notin \mathcal{W}^{(r-1)}$, or $x \in \mathcal{W}^{(r-1)}$ but x is not of k–majority for $\mathcal{W}^{(r-1)}$.

As a consequence of Lemma 2, it follows that $\forall x \in \mathcal{W}_s^{(j)}$

$$\sum_{s=1}^{q} \bar{m}_{\mathcal{W}_s^{(j)}}(x) \leq \left\lfloor \frac{\sum_{s=1}^{q} |\mathcal{W}_s^{(j)}|}{k} \right\rfloor . \tag{19}$$

Taking into account that $m_{\mathcal{W}_s^{(j)}}(x) \leq \bar{m}_{W_s^{(j)}}(x)$, we get

$$\sum_{s=1}^{q} m_{\mathcal{W}_s^{(j)}}(x) \leq \left\lfloor \frac{\sum_{s=1}^{q} |\mathcal{W}_s^{(j)}|}{k} \right\rfloor . \tag{20}$$

Therefore, due to (13), we obtain

$$\sum_{s=1}^{q} m_{\mathcal{W}_s^{(j)}}(x) \leq \left\lfloor \frac{\sum_{i=1}^{2q} |\mathcal{S}_i^{(j)}|}{k} \right\rfloor - \sum_{i=1}^{2q} \mu_{\mathcal{S}_i^{(j)}}, \tag{21}$$

which, by using (16), leads to

$$\sum_{i=1}^{2q} \bar{m}_{\mathcal{S}_i^{(j)}}(x) = \sum_{s=1}^{q} m_{\mathcal{W}_s^{(j)}}(x) + \sum_{i=1}^{2q} \mu_{\mathcal{S}_i^{(j)}} \leq \left\lfloor \frac{\sum_{i=1}^{2q} |\mathcal{S}_i^{(j)}|}{k} \right\rfloor \tag{22}$$

Since $\biguplus_{s=1}^{q} \mathcal{W}_s^{(j-1)} = \biguplus_{i=1}^{q} \mathcal{S}_i^{(j)}$ (here $q = \frac{p}{2^j}$), we deduce that

$$\sum_{s=1}^{q} \bar{m}_{\mathcal{W}_s^{(j-1)}}(x) \leq \left\lfloor \frac{\sum_{s=1}^{q} |\mathcal{W}_s^{(j-1)}|}{k} \right\rfloor . \tag{23}$$

Therefore, we have proved that, since eq. (19) holds in the j–th step, then (as a consequence of eq. (23)) still holds in the $(j - 1)$–th step. For $j = 0$, eq. (20) becomes:

$$\sum_{i=1}^{p} m_{\mathcal{A}_i}(x) \leq \sum_{i=1}^{p} \bar{m}_{\mathcal{S}_i^{(0)}}(x) \leq \left\lfloor \frac{\sum_{i=1}^{p} |\mathcal{A}_i|}{k} \right\rfloor . \tag{24}$$

Consequently, we are led to a contradiction, since x is not a k–majority element for $\biguplus_{i=1}^{p} \mathcal{A}_i = \mathcal{A}$.

4 Parallel Complexity

In this Section we discuss the parallel complexity of the proposed parallel Space Saving algorithm. At the beginning of the algorithm, the workload is balanced using a block distribution; this is done in steps 1–2 with two simple assignments; therefore, the complexity of the initial domain decomposition is $O(1)$. Next, we determine local candidates in each subarray using the sequential Space Saving algorithm. Owing to the block distribution and to the fact that Space Saving is linear in the number of input elements, the complexity of step 3 is $O(n/p)$. Then, we engage in a parallel reduction in step 4 to determine the global candidates for the whole input array. The reduction step requires in the worst case $O(k^2 \ \log \ p)$. Indeed, step 1 of Fig. 2 requires at most $O(k)$ to initialize the *streamSummary* data structure, and steps 2–3 require $O(1)$ time owing to the fact that the minimum value is, by construction, always stored in the first counter of the *streamSummary* data structure. The loops of steps 4–6 and 7–9 require at most $O(k^2)$. This is because incrementing a counter in the StreamSummaryUpdate Algorithm of Fig. 3 requires in our case $O(k)$ time in the worst case instead of constant time. The reason is that, in order to increment one-shot the counter by *count* ≥ 1 occurrences, we may need to scan (in the worst case) all of the buckets in the *streamSummary* data structure to find the correct place to insert a new bucket if needed. This is because the counters must be always maintained in sorted order. All of the other steps of StreamSummaryUpdate require $O(1)$ time in the worst case. In step 10 we simply put the data stored in the *streamSummary* in the *output* array; this requires at most $O(k)$. Finally, we update the output array so that the ϵ and ϵ_p fields correctly reflect the fact that the items have been updated subtracting the minimum values in both the input arrays (steps 12–13) and return *output* as the result of the reduction step. To recap, since we do $O(k^2)$ work in each step of the parallel reduction and there are $\log \ p$ such steps, the overall complexity of the reduction is $O(k^2 \ \log \ p)$. It follows that the overall complexity of the parallel Space Saving algorithm is $O(n/p + k^2 \ \log \ p)$. When $k = O(1)$, i.e., in all of the cases of practical interest, the parallel complexity is $O(n/p + \log \ p)$. We are now in the position to state the following Theorem:

Theorem 3 *The algorithm is cost–optimal for $k = O(1)$.*

Proof Cost–optimality requires by definition that asymptotically $pT_p = T_1$ where T_1 represents the time spent on one processor (sequential time) and T_p the time spent on p processors. The sequential algorithm requires $O(n)$ in the worst case, and the parallel complexity of our algorithm is $O(n/p + \log \ p)$ when $k = O(1)$. It follows from the definition that the algorithm is cost–optimal for $n = \Omega(p \log p)$.

We proceed with the analysis of isoefficiency and scalability. The sequential algorithm has complexity $O(n)$; the parallel overhead is $T_o = pT_p - T_1$. In our case, $T_o = p(n/p + \log \ p) - n = p \log \ p$. The isoefficiency relation [28] is then $n \geq p \ \log \ p)$. Finally, we derive the scalability function of this parallel system [46]. This function shows how memory usage per processor must grow to maintain efficiency at a desired level. If the isoefficiency relation is $n \geq f(p)$ and $M(n)$ denotes the amount of memory required for a problem of size n, then $M(f(p))/p$ shows how memory usage per processor must increase to maintain the same level of efficiency. Indeed, in order to maintain efficiency when increasing p, we must increase n as well, but on parallel computers the maximum problem size is limited by the available memory, which is linear in p. Therefore, when the scalability function $M(f(p))/p$ is a constant C, the parallel algorithm is perfectly scalable; Cp represents instead the limit for scalable algorithms. Beyond this point an algorithm is not scalable (from this point of view). In our case the function describing how much memory is used for a problem of size n is given by $M(n) = n$. Therefore, $M(f(p))/p = O(\log\ p)$ with $f(p)$ given by the isoefficiency relation.

5 The Riemann–Hurwitz and Zipf–Mandelbrot Distributions and Complex Networks: Numerical Simulations

In this Section we report the experimental results we have obtained running the parallel Space Saving algorithm on on an IBM cluster consisting of 30 IBM p575 nodes. Each SMP node is configured with 16 4.7 Ghz dual core Power 6 CPUs with 32 MB level 3 cache and 128 GB of main memory. Two SMT (Simultaneous Multi–Threading) threads per core are enabled, so that each node provides 64 virtual cores (32 physical cores). The interconnection network is Infiniband 4x DDR with 1.19 μs MPI latency and 10 Gbps bandwidth (20 Gbps bidirectional). Our parallel implementation, developed in C using MPI, is based on the sequential source code for the *Space Saving* algorithm developed in [17].

The input distribution used in our experiments is the Riemann–Hurwitz's one (Hurwitz for short), and its particular case, the Zipf one, which is one of the most used in experiments related to sequential algorithms for frequent items.
We recall that the Zipf distribution has probability density function

$$P_Z(x) = \frac{x^{-(\rho+1)}}{\zeta(\rho+1)} \quad x \geq 1, \tag{25}$$

where ρ is a positive real parameter controlling the skew of the distribution and

$$\zeta = \sum_{k=1}^{\infty} \frac{1}{k^s}, \qquad \text{Re s} > 1 \tag{26}$$

is the Riemann zeta function [30]. The Hurwitz distribution has p.d.f.

$$P_H(x) = \frac{x^{-(\rho+1)}}{\zeta(\rho+1,a)} \quad x \geq 1, \tag{27}$$

where

$$\zeta = \sum_{k=1}^{\infty} \frac{1}{(k+a)^s}, \qquad \text{Re s} > 1, \quad \text{Re a} > 0. \tag{28}$$

is the Riemann–Hurwitz zeta function [30]. We have designed and carried out some experiments characterized by the following parameters: the input size n, k and ρ.

The main motivation to use such distributions is that both of them play a crucial role in the rapidly–growing theory of *complex networks* [42], [3], [7].

We recall that a network is essentially a *graph*, i.e. a set of *vertices* or *nodes*, connected by means of *edges* [8]. Many examples of networks arise in applied sciences, e.g. the world wide web, social networks, neural networks, distribution networks, etc. If i is a selected node in a given network, we denote by k_i the number of edges connecting the node i with other k_i nodes. Usually, in a realistic network different nodes may have a different number of edges originating from them. This number is called the *node degree*. A random graph is a graph in which the probability that a randomly selected node has exactly k edges is described by a probability distribution function (p. d. f.) $P(k)$.

The simplest case is the Poisson random graph, introduced by Erdös and Rényi [22]. As has been discovered in [5], an important class of large networks are scale–free, i.e. their degree distribution follows a power law for large k. A huge literature exists now on scale–free models, due to their interesting properties, like growth and preferential attachment (see [3]). Another reason to study these models is that they are intimately connected with nonextensive statistical mechanics ([50]). In the theory of scale–free networks, the p.d.f. emerging is the Hurwitz one, or its simplified version, the Zipfian one. By using a generating function approach, in [43] has been proved that a phase transition occurs, in the Zipfian model, for the value of the skewness ρ such that:

$$\zeta(\rho - 1) = \zeta(\rho), \tag{29}$$

which gives the critical value $\rho_c = 2.4788\ldots$ The transition consists in the fact that, below this value, a giant component of the graph exists; above it, there is no giant component. In our numerical simulations, we have selected a range for ρ including ρ_c. Also, we have fixed $a = 0.5$ in all simulations involving the Hurwitz distribution. For each input distribution generated, the algorithm has been run ten times on up to 8 cores (one core per node), and the results have been averaged for each number of cores, over all of the runs. The input elements are 32 bits unsigned integers. Table 1 reports the values actually used in each of the experiments.

Fig. 4 presents the predicted running times versus the experimental ones. We have plotted experimental versus predicted running times for both the fastest and the slowest experiments, characterized respectively by the parameters $k = 100, \rho = 3.0, n = 4,000,000,000$ and $k = 10, \rho = 1.5, n = 8,000,000,000$. Our model for predicting the running time is the following:

$$R(p, n, k) = \chi n/p + \log(p)(\lambda + 16k/\beta), \tag{30}$$

where χ is the time for processing an input item, whilst λ and β are respectively the latency and bandwidth of the interconnection network. The parameter χ is obtained by executing on one core (sequential run) the algorithm and dividing the input size n by the obtained running time. For the Infiniband interconnection network installed on our machine, $\lambda = 1.19 \mu s$ and $\beta = (10/8) * 1024^3 = 1342177280$ $bytes/s$. The constant 16 in the bandwidth term models the communication of 4 (two pairs of candidate and corresponding weight) arrays, in which each element is an unsigned integer represented using 4 bytes. This constant is multiplied by k, which is the length of the arrays.

As shown, the model correctly approximates the running time of the algorithm.

Fig. 5–6 present the actual performances we obtained running the experiments of Table 1 respectively on Zipfian and Hurwitz distributed elements. It is worth noting here that the obtained results confirm the theoretical cost–optimality of the proposed algorithm, with speedup and efficiency values respectively greater than or equal to p and 1 in all of the cases, except for the Zipfian experiment characterized by the parameters $k = 100, \rho = 3.0, n = 8,000,000,000$. However, even in this case the efficiency is still 0.98, an extremely high value. It is worth noting here that the slightly superlinear speedup observed experimentally is due to the IBM p475 memory hierarchy and to related cache effects. So-called superlinear speedups, i.e., speedups which are greater than the number of processors, are a source of confusion because in theory this phenomenon is not possible according to Brent's principle [11] (which states that a single processor can simulate a p-processor algorithm with a uniform slowdown factor of p).

Experimentally, a superlinear speedup can be observed without violating Brents principle when the storage space required to run the code on a particular instance exceeds the memory available on the single-processor machine, but not that of the parallel machine used for the simulation. In such a case, the sequential code needs to swap to secondary memory (disk) while the parallel code does not, therefore yielding a dramatic slowdown of the sequential code. On a more modest scale, the same problem could occur one level higher in the memory hierarchy, with the sequential code constantly cache-faulting while the parallel code can keep all of the required data in its cache subsystems. A sequential algorithm using M bytes of memory will use only M/p bytes on each processor of a p processor parallel system, so that it is easier to keep all of the data in cache memory on the parallel machine. This is exactly what happened in our simulations.

We recall here that other possible sources of superlinear speedup include some brute–force search problems and the use of a suboptimal sequential algorithm. A parallel system might exhibit such behavior in search algorithms. In search problems performed by exhaustively looking for the solution, suppose

Table 1: Design of experiments for Zipfian and Hurwitz distributions

experiment	n (billions)	k	ρ
1	4	10	1.5
2	4	10	3.0
3	4	100	1.5
4	4	100	3.0
5	8	10	1.5
6	8	10	3.0
7	8	100	1.5
8	8	100	3.0

the solution space is divided among the processors for each one to perform an independent search. In a sequential implementation the different search spaces are attacked one after the other, while in parallel they can be done simultaneously, and one processor may find the solution almost immediately, yelding a superlinear speedup. A parallel system might also exhibit such behavior when using a suboptimal sequential algorithm: each processing element spends less than the time required by the sequential algorithm divided by p solving the problem. Generally, if a purely deterministic parallel algorithm were to achieve better than p times the speedup over the current sequential algorithm, the parallel algorithm (by Brent's principle) could be emulated on a single processor one parallel part after another, to achieve a faster serial program, which contradicts the assumption of an optimal serial program.

We now discuss the parallel mean errors we obtained running the experiments, depicted in Fig. 7. For each experiment carried out, we plot the average of the ϵ error computed over all of the counters. We note immediately that the errors were zero in all of the experiments but the ones characterized by the parameters $k = 10, \rho = 1.5$. The errors are higher for the Hurwitz case. Anyway, the order of magnitude of the mean errors ranges from 10^{-10} to 10^{-9} and is therefore quite low. Finally, we recall here that the error p_ϵ associated to a counter in the output can be used to estimate a confidence interval for the number of actual occurrences of the element tracked by the counter: given an hypothetical element x with o occurrences reported by its counter, then the actual occurrences are in the range $[o - p_\epsilon, o]$. For instance, if $x = 113, o = 315, p_\epsilon = 5$, then the confidence interval is [310, 315].

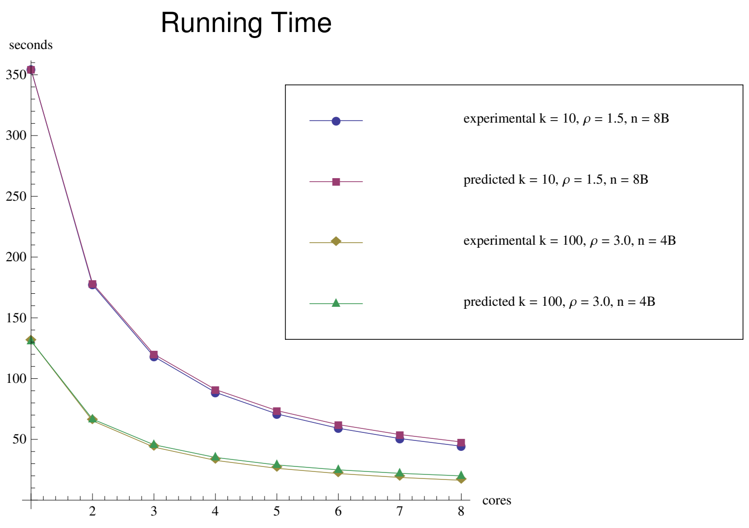
Acknowledgements The research of M. Cafaro has been supported by CMCC, Italy. The research of P. Tempesta has been supported by the grant FIS2008–00260, Ministerio de Ciencia e Innovación, Spain. The authors would like to thank G. Cormode and M. Hadjieleftheriou for making freely available their sequential implementation of the Space Saving algorithm.

References

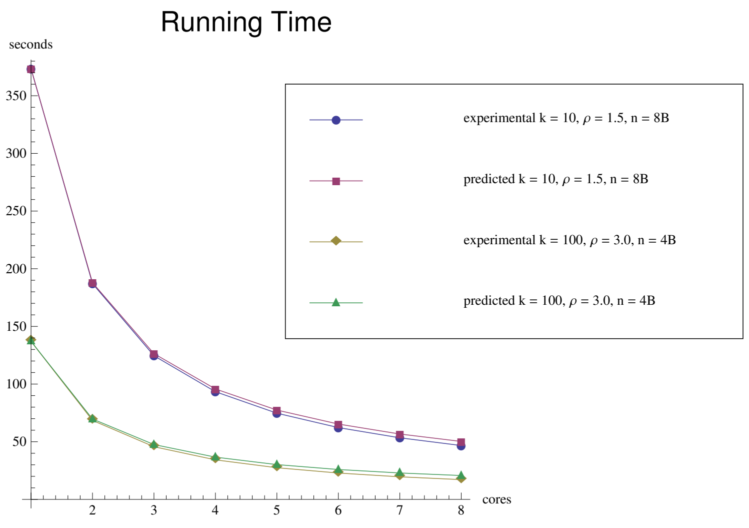
1. *Computational Linguistics and Intelligent Text Processing, 7th International Conference, CICLing 2006, Mexico City, Mexico, February 19-25, 2006, LNCS 3878.* Springer–Verlag, 2006.
2. G. M. Adel'son–Vel'skii and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1262, 1962.
3. R. Albert and A.-L. Barabási. Statistical mechanics of complex networks. *Rev. Mod. Phys.*, 74:47–97, June 2002.
4. B. Babcock and C. Olston. Distributed top-k monitoring. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 28–39. ACM, 2003.
5. A.-L. Barabási and R. Albert. Emergence of Scaling in Random Networks. *Science*, 286(5439):509–512, Oct. 1999.
6. K. Beyer and R. Ramakrishnan. Bottom–up computation of sparse and iceberg cubes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data. ACM, New York*, pages 359–370, 1999.
7. S. Boccaletti, V. Latora, Y. Moreno, M. Chavez, and D. Hwang. Complex networks: Structure and dynamics. *Physics Reports*, 424(4-5):175–308, Feb. 2006.
8. B. Bollobás. *Random Graphs.* Cambridge University Press, 2 edition, Jan. 2001.
9. R. Boyer and J. Moore. Mjrty – a fast majority vote algorithm. Technical Report 32, Institute for Computing Science, University of Texas, Austin, 1981.
10. R. Boyer and J. S. Moore. Mjrty – a fast majority vote algorithm. In *Automated Reasoning: Essays in Honor of Woody Bledsoe, Automated Reasoning Series, Kluwer Academic Publishers, Dordrecht, The Netherlands*, pages 105–117. 1991.
11. R. P. Brent. The Parallel Evaluation of General Arithmetic Expressions. *J. ACM*, 21(2):201–206, 1974.
12. S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 255–264, New York, NY, USA, 1997. ACM.
13. M. Cafaro and P. Tempesta. Finding frequent items in parallel. *Concurrency and Computation: Practice and Experience, to appear.*
14. M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *ICALP '02: Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, pages 693–703, London, UK, 2002. Springer–Verlag.
15. G. Cormode and M. Garofalakis. Sketching streams through the net: distributed approximate query tracking. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 13–24. VLDB Endowment, 2005.
16. G. Cormode and M. Garofalakis. Approximate continuous querying over distributed streams. *ACM Trans. Database Syst.*, 33:9:1–9:39, June 2008.
17. G. Cormode and M. Hadjieleftheriou. Finding frequent items in data streams. *Proc. VLDB Endow.*, 1:1530–1541, August 2008.
18. G. Cormode and M. Hadjieleftheriou. Finding the frequent items in streams of data. *Commun. ACM*, 52(10):97–105, 2009.
19. G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.
20. S. Das, S. Antony, D. Agrawal, and A. E. Abbadi. Thread cooperation in multicore architectures for frequency counting over multiple data streams. *Proc. VLDB Endow.*, 2(1):217–228, 2009.
21. E. D. Demaine, A. López-Ortiz, and J. I. Munro. Frequency estimation of internet packet streams with limited space. In *ESA*, pages 348–360, 2002.
22. P. Erdös and A. Rényi. On random graphs, I. *Publicationes Mathematicae (Debrecen)*, 6:290–297, 1959.
23. C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *IMW '01: Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pages 75–80, New York, NY, USA, 2001. ACM.

24. M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing iceberg queries efficiently. In *Proceedings of the 24th International Conference on Very Large Data Bases, VLDB. Morgan–Kaufmann, San Mateo, Calif.*, pages 299–310, 1998.
25. M. Fischer and S. Salzberg. Finding a majority among n votes: Solution to problem 81–5. *J. of Algorithms*, (3):376–379, 1982.
26. R. Fuller and M. M. Kantardzic. Distributed monitoring of frequent items. *Trans. MLDM*, 1(2):67–82, 2008.
27. P. B. Gibbons and Y. Matias. Synopsis data structures for massive data sets. In *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science: Special Issue on External Memory Algorithms and Visualization, vol. A.*, pages 39–70, Boston, MA, USA, 1999. American Mathematical Society.
28. A. Grama, A. Gupta, and V. Kumar. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Parallel and Distributed Technology*, 1(3):12–21, Aug. 1993.
29. R. Y. S. Hung, L.-K. Lee, and H. F. Ting. Finding frequent items over sliding windows with constant update time. *Inf. Process. Lett.*, 110:257–260, March 2010.
30. H. Iwaniec and E. Kowalski. *Analytic Number Theory.* Number 53 in Amer. Math. Soc. Colloq. Publ. Amer. Math. Soc., Providence, RI, 2004.
31. R. M. Karp, S. Shenker, and C. H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, 28(1):51–55, 2003.
32. R. Keralapura, G. Cormode, and J. Ramamirtham. Communication-efficient distributed monitoring of thresholded counts. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 289–300. ACM, 2006.
33. B. Lahiri and S. Tirthapura. Identifying frequent items in a network using gossip. *J. Parallel Distrib. Comput.*, 70(12):1241–1253, 2010.
34. M. Li and W.-C. Lee. Identifying frequent items in p2p systems. In *Distributed Computing Systems, 2008. ICDCS '08. The 28th International Conference on*, pages 36 –44, june 2008.
35. B. Mandelbrot. Information theory and psycholinguistics: A theory of word frequencies. In *Language: selected readings, edited by R.C. Oldfield and J.C. Marchall, Penguin Books.* 1968.
36. N. Manerikar and T. Palpanas. Frequent items in streaming data: An experimental evaluation of the state-of-the-art. *Data Knowl. Eng.*, 68(4):415–430, 2009.
37. A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston. Finding (recently) frequent items in distributed data streams. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 767–778. IEEE Computer Society, 2005.
38. G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *In VLDB*, pages 346–357, 2002.
39. A. Metwally, D. Agrawal, and A. E. Abbadi. Efficient computation of frequent and top-k elements in data streams. In *International Conference on Database Theory*, pages 398–412, 2005.
40. J. Misra and D. Gries. Finding repeated elements. *Sci. Comput. Program.*, 2(2):143–152, 1982.
41. D. Mouillot and A. Lepretre. Introduction of relative abundance distribution (rad) indices, estimated from the rank-frequency diagrams (rfd), to assess changes in community diversity. *Environmental Monitoring and Assessment*, 63(2):279–295, 2000.
42. M. E. J. Newman. The Structure and Function of Complex Networks. *SIAM Review*, 45(2):167–256, 2003.
43. M. E. J. Newman, S. H. Strogatz, and D. J. Watts. Random graphs with arbitrary degree distributions and their applications. *Physical Review E*, 64(2):026118+, July 2001.
44. N. Ntarmos, P. Triantafillou, and G. Weikum. Distributed hash sketches: Scalable, efficient, and accurate cardinality estimation for distributed multisets. *ACM Trans. Comput. Syst.*, 27:2:1–2:53, February 2009.
45. R. Pan, L. Breslau, B. Prabhakar, and S. Shenker. Approximate fairness through differential dropping. *SIGCOMM Comput. Commun. Rev.*, 33(2):23–39, 2003.
46. M. J. Quinn. *Parallel Programming in C with MPI and OpenMP.* McGraw–Hill, June 2003.

47. A. Syropoulos. Mathematics of multisets. In *In Multiset Processing: Mathematical, computer science, and molecular computing points of view, LNCS 2235*, pages 347–358. Springer–Verlag, 2001.
48. J. Teubner, R. Muller, and G. Alonso. Frequent item computation on a chip. *Knowledge and Data Engineering, IEEE Transactions on*, 23(8):1169 –1181, aug. 2011.
49. H. Thanh Lam and T. Calders. Mining top-k frequent items in a data stream with flexible sliding windows. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '10, pages 283–292, New York, NY, USA, 2010. ACM.
50. S. Thurner, F. Kyriakopoulos, and C. Tsallis. Unified model for network dynamics exhibiting nonextensive statistics. *Physical Review E*, 76(3):036111+, Sept. 2007.
51. S. Wang, X. Hao, H. Xu, and Y. Hu. Mining frequent items based on bloom filter. In *Proceedings of the Fourth International Conference on Fuzzy Systems and Knowledge Discovery - Volume 04*, FSKD '07, pages 679–683, Washington, DC, USA, 2007. IEEE Computer Society.
52. G. K. Zipf. *Selected Studies of the Principle of Relative Frequency in Language*. Cambridge University Press, 1932.

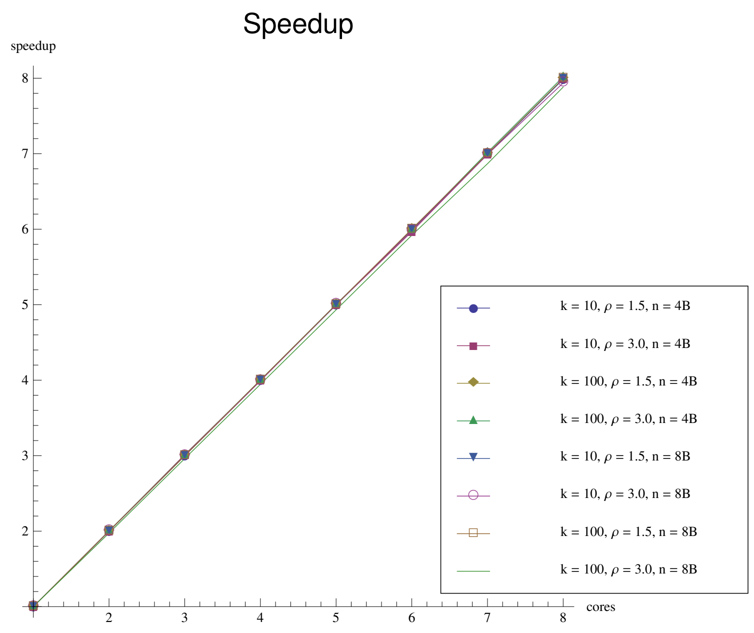


(a) Zipfian

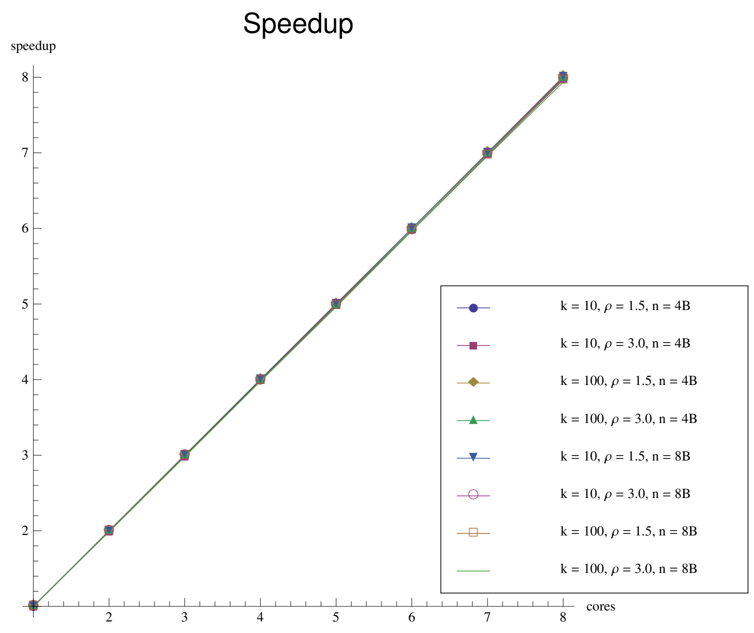


(b) Hurwitz

Fig. 4: Prediction model

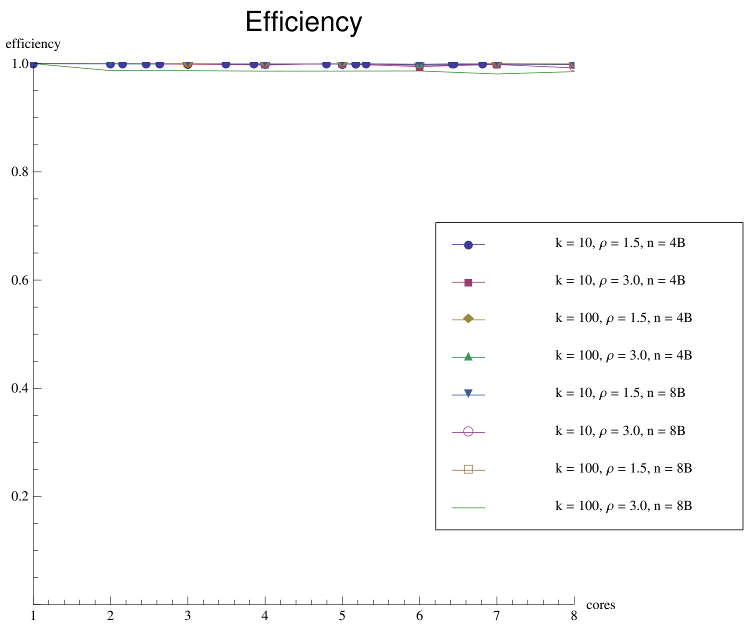


(a) Zipfian

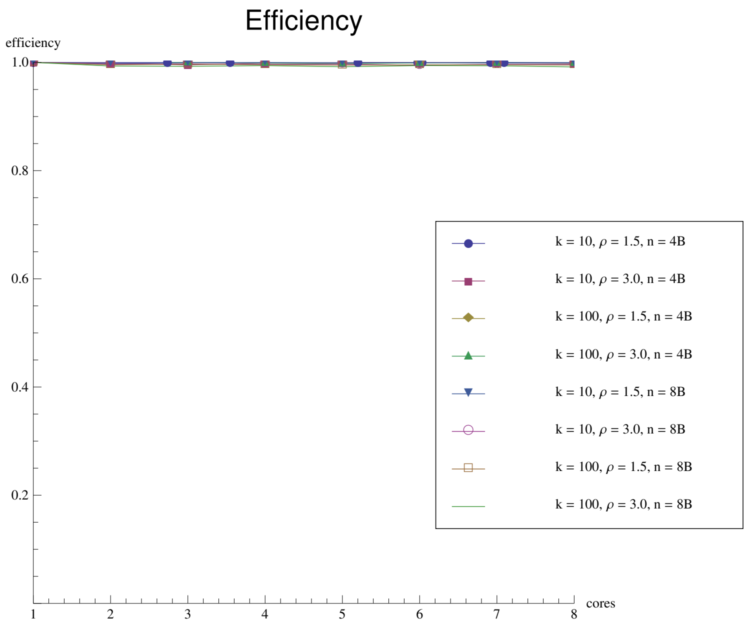


(b) Hurwitz

Fig. 5: Speedup

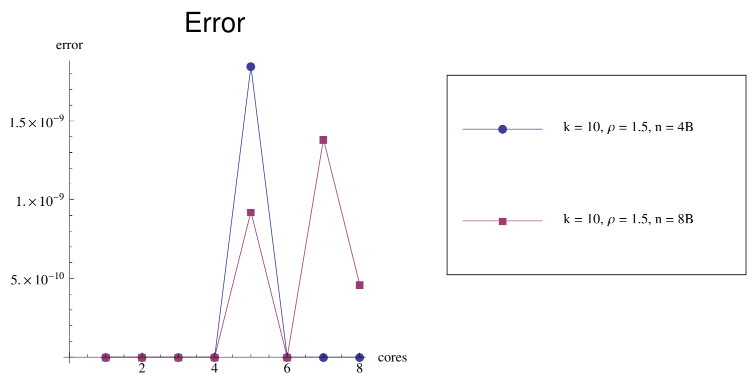


(a) Zipfian

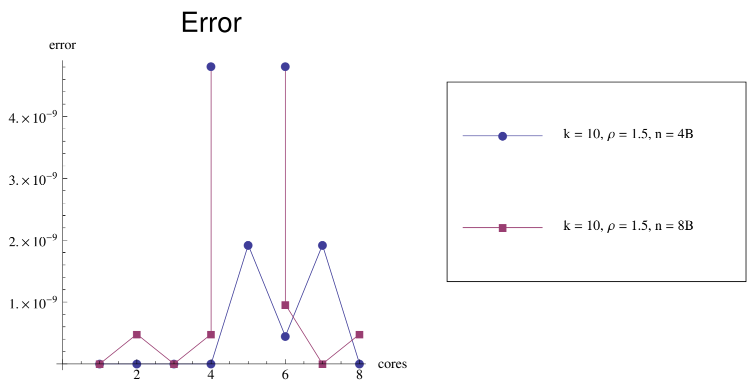


(b) Hurwitz

Fig. 6: Efficiency



(a) Zipfian



(b) Hurwitz

Fig. 7: Errors