Parallel Mining of Correlated Heavy Hitters

Marco Pulimeno[1], Italo Epicoco[1,2], Massimo Cafaro[1,2](✉), Catiuscia Melle[1], and Giovanni Aloisio[1,2]

[1] University of Salento, 73100 Lecce, Italy
{marco.pulimeno,italo.epicoco,massimo.cafaro,catiuscia.melle, giovanni.aloisio}@unisalento.it
[2] Euro-Mediterranean Center on Climate Changes Foundation, 73100 Lecce, Italy

Abstract. We present a message-passing based parallel algorithm for mining Correlated Heavy Hitters from a two-dimensional data stream. To the best of our knowledge, this is the first parallel algorithm solving the problem. We show, through experimental results, that our algorithm provides very good scalability, whilst retaining the accuracy of its sequential counterpart.

Keywords: Correlated Heavy Hitters · Message-passing

1 Introduction

Mining of Correlated Heavy Hitters (CHH) is a problem which has been recently proposed by [22]. Determining CHHs is a data mining task which commonly arises in the context of network monitoring and management, as well as anomaly and intrusion detection.

When considering the stream of pairs (source address, destination address) consisting of IP packets passing through a router, it is useful and important being able to identify the nodes responsible for the majority of the traffic passing through that router (i.e., the frequent items over a single dimension); however, for each given frequent source, we are also interested to discover the target destinations receiving the majority of connections by the same source.

Therefore, the mining process works as follows: initially the most important sources are detected as frequent items over the first dimension, then we mine the frequent destinations in the context of each identified source, i.e., the stream's correlated heavy hitters. We recall here preliminary notations and definitions, that shall be used in order to formally state the problem we solve in this paper through a message-passing based parallel algorithm. Let σ be a stream of tuples (x, y) of length n: $\sigma =< (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n) >$. The frequency f_{xy} of the tuple (x, y) is defined as follows.

Definition 1. *The frequency f_{xy} of the tuple (x, y) in the stream σ is given by $f_{xy} = |\{i : (x = x_i) \wedge (y = y_i)\}|$.*

© Springer International Publishing AG, part of Springer Nature 2018
O. Gervasi et al. (Eds.): ICCSA 2018, LNCS 10964, pp. 627–641, 2018.
https://doi.org/10.1007/978-3-319-95174-4_48

If we consider the sub-stream induced by the projection of the tuples of the stream σ on its first dimension, also referred to as the *primary dimension*, we can define the frequency f_x of an item appearing as first element in a tuple (x, y).

Definition 2. *The frequency f_x of an item x which appears as first element in a tuple (x, y) in the stream σ is given by $f_x = |\{i : (x = x_i)\}|$.*

The items x appearing in the primary dimension of the stream σ are referred to as *primary items*. Similarly, the items y appearing in the *secondary dimension* are referred to as *secondary items* or as *correlated items*.

The Exact Correlated Heavy Hitters problem can not be solved using limited space and only one pass through the input stream, hence the Approximate Correlated Heavy Hitters problem (ACHH) is introduced [22]. We state the problem as follows.

Problem 1. Approximate Correlated Heavy Hitters problem (ACHH problem).

Given a data stream σ of length n consisting of (x, y) tuples in which the item x is drawn from a universe set $\mathcal{U}_1 = \{1, \ldots, m_1\}$ and the item y is drawn from a universe set $\mathcal{U}_2 = \{1, \ldots, m_2\}$, two user-defined thresholds ϕ_1 and ϕ_2 such that $0 < \phi_1 < 1$ and $0 < \phi_2 < 1$ and two error bounds ϵ_1 and ϵ_2 such that $0 < \epsilon_1 < \phi_1$ and $0 < \epsilon_2 < \phi_2$, the Approximate Correlated Heavy Hitters (ACHH) problem requires determining all of the primary items x such that

$$f_x > \phi_1 n \tag{1}$$

and no items with

$$f_x \leq (\phi_1 - \epsilon_1)n \tag{2}$$

should be reported; moreover, we are required to determine for each frequent primary candidate x, all of the tuples (x, y) such that

$$f_{xy} > \phi_2 f_x \tag{3}$$

and no tuple (x, y) such that

$$f_{xy} \leq (\phi_2 - \epsilon_2) f_x \tag{4}$$

should be reported.

In this paper, we propose a parallel algorithm for solving the Approximate Correlated Heavy Hitters problem. The paper is organized as follows. In Sect. 2 we recall related work. We present our parallel algorithm in Sect. 3, and thoroughly analyze it in Sect. 4. Extensive experimental results are provided in Sect. 5. Finally, we draw our conclusions in Sect. 6.

2 Related Work

Mining frequent items is one of the most important topics in data mining. As such, it has attracted a number of researchers, who have published extensively on this subject. Here, we recall the most important work.

Misra and Gries [25] generalized the seminal work of Boyer and Moore [1,2] (the so called MJRTY algorithm). Their algorithm behaves exactly as MJRTY, but uses multiple counters, i.e. pairs (item, frequency), to keep track of the frequent items in the input stream. Interestingly, this algorithm, has been forgotten for about twenty years, and later rediscovered and slightly improved with regard to speed (by using a clever summary data structure) by Demaine et al. [16] (the so-called *Frequent* algorithm) and Karp et al. [21].

Counters are also used in many other algorithms, including *Sticky Sampling*, *Lossy Counting* [23], and *Space Saving* [24]. It is worth noting here that Space Saving is still the most accurate algorithm for mining frequent items.

A different class of algorithms is based on the use of a sketch data structure, which is usually a bi-dimensional array of counters. Items are mapped, through a set of hash functions (one for each row of the sketch), to corresponding sketch cells. Each cell holds a counter, whose values is then updated as required by the algorithm. Sketch–based algorithms include *CountSketch* [11], *Group Test* [14], *Count-Min* [13] and *hCount* [20].

Parallel algorithms for frequent items include message-passing, shared-memory and accelerators based algorithms. Almost all of the proposed algorithms are parallel versions of Frequent and Space Saving. Among message-passing algorithms, we recall here [9,10] (slightly improved in [5]). With regard to shared-memory architectures, it is worth citing [15,26,29,30]. Recently, [27] proposed novel shared-memory algorithms. Accelerator based algorithms for frequent items exploiting a GPU (Graphics Processing Unit) or the Intel Phi processor include [3,8,18,19].

Mining time faded frequent items has been investigated in [4,7,12,28]. A parallel message-passing based algorithm has been recently proposed in [6].

Regarding CHHs, an algorithm based on the nested application of Frequent has been recently presented in [22]. The outermost application mines the primary dimension, whilst the innermost one mines correlated secondary items. The main drawbacks of this algorithm, being based on Frequent, are the accuracy (which is very low), the huge amount of space required and the rather slow speed (owing to the nested summaries).

In [17], we proposed a fast and accurate algorithm for mining CHHs. Our Cascading Space Saving Correlated Heavy Hitters (CSSCHH) algorithm exploits the basic ideas of Space Saving, combining two summaries for tracking the primary item frequencies and the tuple frequencies. We refer to our algorithm as Cascading Space Saving since it is based on the use of two distinct and independent applications of Space Saving.

A stream summary $\mathcal{S}$ with k counters is the data structure used by Space Saving in order to monitor up to k distinct items. Space Saving processes one item at a time. When the item is already monitored by a counter, its estimated

frequency is incremented by one. When it is not monitored, there are two possibilities. If a counter is available, it will be in charge of monitoring the item and its estimated frequency is set to one. Otherwise, if all of the counters are already occupied (their frequencies are different from zero), the counter storing the item with minimum frequency is incremented by one, then the monitored item is replaced by the new item. It can be proved that Space Saving correctly reports in its summary all of the ϕ-frequent items of the processed input stream with $\phi > \frac{1}{k}$ [24].

Therefore, we use two independent Space Saving stream summaries as data structures. The first, denoted by $\mathcal{S}^p$, and referred to as the primary stream summary, monitors a subset of primary items which appears in the stream through the use of k_1 distinct counters. The second, denoted by $\mathcal{S}^t$, includes k_2 counters and monitors a subset of the tuples which appear in the stream.

We proved that CSSCHH is correct and outperforms the algorithm proposed in [22] with regard to speed, accuracy and space required; we also showed how to select the values of k_1 and k_2 in order to minimize the space required. Full details can be found in [17]. In this work we design a parallel version of CSSCHH which we call PCSSCHH. To the best of our knowledge, this is the first parallel algorithm for message-passing architectures solving the ACHH problem.

3 A Parallel Algorithm for the ACHH Problem

In this paper we assume an offline setting, in which the stream tuples have been stored as a static dataset. This is not restrictive, since we shall show that our algorithm can also work in the streaming (online) setting as well. In the offline setting, we partition the input dataset by using a simple 1D block-based domain decomposition among the available p processes; then, in parallel, each process updates its local summaries with the items belonging to its own block. Once the blocks have been processed, one of the processes is in charge of determining the CHHs. The processes engage in a parallel reduction in which their summaries are merged into global summaries preserving all of the information stored in the original local summaries. These summaries can then be queried to return the CHHs.

The streaming (online) setting is related to a distributed scenario in which there are p distributed sites, each handling a different stream $\sigma_i, i = 1, \ldots, p$. One of the p sites may act as a centralized coordinator, or there can be another different site taking this responsibility. The coordinator broadcasts, when required, a "query" message to the p sites, which then temporarily stop processing their sub-streams, and engage in the merge procedure. The distributed sites can be thought as being multi-threaded processes, in which one thread processing the stream temporarily stops when a query message is received from the coordinator, creates a copy of its local summaries and then resumes stream processing whilst another thread engages in the distributed merging using the copies of the summaries.

Our PCSSCHH algorithm starts by initializing the $\mathcal{S}^p$ primary stream summary data structure allocating k_1 counters and the correlated $\mathcal{S}^t$ stream summary allocating k_2 counters. Algorithm 1 presents the pseudocode related to the initialization phase of PCSSCHH.

Algorithm 1. PCSSCHH Init

Require: Threshold for primary items ϕ_1; threshold for correlated items ϕ_2; tolerance for primary items ϵ_1; tolerance for correlated items ϵ_2.
Ensure: Properly initialized $\mathcal{S}^p$ and $\mathcal{S}^t$ stream summaries

1: **procedure** PCSSCHH-Init($\phi_1, \phi_2, \epsilon_1, \epsilon_2$)
2: $\quad\beta \leftarrow \frac{1}{\epsilon_2 \phi_1}$
3: $\quad\gamma \leftarrow \frac{\epsilon_2 + \phi_2}{\epsilon_2 \phi_1}$
4: $\quad k_1 \leftarrow \max\left\{\frac{1}{\epsilon_1}, \gamma + \sqrt{\beta\gamma}\right\}$
5: $\quad k_2 \leftarrow \beta \frac{k_1}{k_1 - \gamma}$
6: $\quad$Allocate k_1 counters for $\mathcal{S}^p$
7: $\quad$Allocate k_2 counters for $\mathcal{S}^t$
8: $\quad$**return** $\mathcal{S}^p$ and $\mathcal{S}^t$
9: **end procedure**

Once a processor has initialized its summaries, it can begin processing the stream's tuples. The n tuples of the stream σ are distributed, through domain decomposition, to the p processors so that each one is responsible for either $\lfloor n/p \rfloor$ or $\lceil n/p \rceil$ tuples; let $left$ and $right$ be respectively the indices of the first and last tuple handled by the process with rank id (ranks are numbered from 0 to $p-1$), then:

$$left = \lfloor (id - 1)\ n/p \rfloor; \tag{5}$$

$$right = \lfloor id\ n/p \rfloor - 1. \tag{6}$$

PCSSCHH is presented in pseudocode as Algorithms 2 and 3. Each processor starts processing its own substream, which consists of all of the tuples from *left* to *right*. In particular, items belonging to the primary dimension are mined using the $\mathcal{S}^p$ summary, whilst tuples are mined using the $\mathcal{S}^t$ summary. Next, parallel reductions based on the COMBINE user's defined reduction operator provide

Algorithm 2. PCSSCHH

Require: σ: a stream consisting of tuples (x, y); $\mathcal{S}^p$ and $\mathcal{S}^t$: stream summaries;
Ensure: $\mathcal{S}^p_g$ and $\mathcal{S}^t_g$ stream summaries.

1: **procedure** PCSSCHH-Update($\mathcal{S}^p, \mathcal{S}^t, \sigma$)
2: $\quad$SpaceSaving($\mathcal{S}^p$, $left$, $right$)
3: $\quad$SpaceSaving($\mathcal{S}^t$, $left$, $right$)
4: $\quad\mathcal{S}^p_g \leftarrow$ ParallelReduction($\mathcal{S}^p$, k_1, COMBINE)
5: $\quad\mathcal{S}^t_g \leftarrow$ ParallelReduction($\mathcal{S}^t$, k_2, COMBINE)
6: **end procedure**

Algorithm 3. COMBINE

Require: $\mathcal{S}_1$, $\mathcal{S}_2$: summaries ordered by counters' frequency; k, number of counters in each summary
Ensure: the *combined summary* $\mathcal{S}_C$

```
 1: procedure COMBINE(S_1, S_2, k)
 2:     m_1 ← S_1[0].f̂                          ▷ minimum of all of the frequencies in S_1
 3:     m_2 ← S_2[0].f̂                          ▷ minimum of all of the frequencies in S_2
 4:     let S_C be an empty summary
 5:     for each counter S_1[j] in S_1 do
 6:         new_counter.i ← S_1[j].i
 7:         counter_{S_2} ← S_2.FIND(S_1[j].i)
 8:         if counter_{S_2} then
 9:             new_counter.f̂ ← S_1[j].f̂ + counter_{S_2}.f̂
10:             S_2.REMOVE(counter_{S_2})
11:         else
12:             new_counter.f̂ ← S_1[j].f̂ + m_2
13:         end if
14:         S_C.PUT(new_counter)
15:     end for
16:     for each counter S_2[j] in S_2 do
17:         new_counter.i ← S_2[j].i
18:         new_counter.f̂ ← S_2[j].f̂ + m_1
19:         S_C.PUT(new_counter)
20:     end for
21:     S_C.PRUNE(k)  ▷ Select k counters with the greatest frequencies and delete the
    others
22:     return S_C
23: end procedure
```

the final $\mathcal{S}_g^p$ and $\mathcal{S}_g^t$ summaries (the subscript g stands for "global"), that can be used for answering queries related to CHHs.

The parallel reduction operator (COMBINE) works as follows. We denote by $\mathcal{S}[j].i$ and $\mathcal{S}[j].\hat{f}$ respectively the item monitored by the jth counter of a summary $\mathcal{S}$ and its corresponding estimated frequency. Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be the two summaries to be merged and k their number of counters. We begin determining m_1 and m_2, the minimum frequencies respectively in the input summary $\mathcal{S}_1$ and $\mathcal{S}_2$. After initializing an empty summary $\mathcal{S}_C$, we scan the counters of $\mathcal{S}_1$. For each counter monitoring an item, we search $\mathcal{S}_2$ for a corresponding counter monitoring the same item. If we find it, we initialize a new counter with this item setting as frequency of the item the sum of the frequencies in the corresponding counters of $\mathcal{S}_1$ and $\mathcal{S}_2$ and delete the counter in $\mathcal{S}_2$. Otherwise, we let the frequency be the sum of the frequency of the item in $\mathcal{S}_1$ and m_2. The new counter is then inserted in $\mathcal{S}_C$. Next, we scan the remaining counters in $\mathcal{S}_2$. Since the counters in $\mathcal{S}_2$ corresponding to items in $\mathcal{S}_1$ have been deleted, for each counter in $\mathcal{S}_2$ we prepare a new counter monitoring that item and set its frequency to the sum of the item's frequency in $\mathcal{S}_2$ and m_1. Finally, if the $\mathcal{S}_C$ summary holds more

than k counters, we retain the first k counters with the greatest frequencies and delete the others.

We have proved in [9] that the above reduction correctly merges two Space Saving summaries, and that the resulting $\mathcal{S}_C$ merged summary is affected by an error which is within the error bound of the original input summaries $\mathcal{S}_1$ and $\mathcal{S}_2$.

The final $\mathcal{S}_g^p$ and $\mathcal{S}_g^t$ stream summaries can be queried for CHHs as follows. The query procedure internally uses two lists, F and C. The former stores primary items and their estimated frequencies $(r, \hat{f}_r)$. The latter stores CHHs $(r, s, \hat{f}_{rs})$ in which r is a primary frequent item candidate, s the correlated frequent item candidate and $\hat{f}_{rs}$ the estimated frequency of the tuple (r, s).

The query algorithm inspects all of the k_1 counters in the $\mathcal{S}_g^p$ stream summary. If the frequency of the jth monitored item is greater than the selection criterion (i.e., $\mathcal{S}_g^p[j].\hat{f} > \phi_1 n$), then we add the monitored item $r = \mathcal{S}_g^p[j].i$ and its estimated frequency $\hat{f}_r = \mathcal{S}_g^p[j].\hat{f}$ to F.

The algorithm inspects now all of the k_2 counters of the $\mathcal{S}_g^t$ stream summary. The monitored items in $\mathcal{S}_g^t$ are the tuples (r, s). We check if the primary item r is a primary frequent item candidate (i.e., if $r \in F$); if this condition is true and the estimated frequency of the jth tuple is greater than the selection criterion (i.e., $\mathcal{S}_g^t[j].\hat{f} > \phi_2(\hat{f}_r - \frac{n}{k_1})$), then the triplet $(r, s, \hat{f}_{rs})$ is added to C.

Taking into account the result in [9] related to the correctness of the merge procedure, it still holds what we proved in [17] and restated in Theorems (1) and (2) with reference to PCSSCHH. Thus the reported sets F and C correctly solve the ACHH problem.

Theorem 1. *The PCSSCHH algorithm reports in the outputted set F all of the primary items x whose exact frequency f_x is greater than the threshold, i.e., $f_x > \phi_1 N$ and no items whose exact frequency is such that $f_x \leq (\phi_1 - \frac{1}{k_1})N$.*

Theorem 2. *All of the tuples (x, y) with the item x reported as primary frequent candidate and with exact frequency f_{xy} greater than the threshold ($f_{xy} > \phi_2 f_x$) are reported in the outputted set C as correlated heavy hitter candidates. No tuple with a primary item x reported as frequent primary candidate and with exact frequency less than $f_{xy} \leq (\phi_2 - \frac{k_2\phi_2 + k_1}{k_2(k_1\phi_1 - 1)})f_x$ is reported as correlated heavy hitter candidate.*

The Query procedure is presented as Algorithm 4.

4 Analysis

Regarding the parallel complexity of the algorithm, its worst case complexity is analyzed as follows. The initialization done in Algorithm 1 requires $O(1)$ constant time. Determining the initial domain decomposition requires $O(1)$ time as well. Algorithm 2 requires $O(n/p)$ time to process, using Space Saving, the tuples in the input block (the primary dimension with the $\mathcal{S}^p$ summary, and tuples with the $\mathcal{S}^t$ summary). Indeed, a block consists of $O(n/p)$ tuples, and Space Saving complexity is linear in the length of the input.

Algorithm 4. PCSSCHH Query

Require: $\mathcal{S}_g^p$ and $\mathcal{S}_g^t$ stream summaries.
Ensure: Set of correlated frequent items C

```
 1: procedure PCSSCHH-Query(S_g^p, S_g^t)
 2:     F ← ∅
 3:     for each counter S_g^p[j] in S_g^p do
 4:         r ← S_g^p[j].i; f̂_r ← S_g^p[j].f̂
 5:         if f̂_r > φ_1 n then
 6:             F ← F ∪ {(r, f̂_r)}
 7:         end if
 8:     end for
 9:     for each S_g^t[j] ∈ S_g^t do
10:         (r, s) ← S_g^t[j].i; f̂_rs ← S_g^t[j].f̂
11:         if r ∈ F ∧ (f̂_rs > φ_2(f̂_r − n/k_1)) then
12:             C ← C ∪ {(r, s, f̂_rs)}
13:         end if
14:     end for
15:     return C
16: end procedure
```

Finally, two parallel reductions determine the output. These two reductions require $O((k_1+k_2)\log p)$ time, since the user's defined reduction operator COMBINE (Algorithm 3) requires $O(k\log p)$ time to merge two summaries of k counters. Indeed, the input summaries can be combined in $O(k)$ time, by using the hash tables in the implementation of the summaries.

For each item in $\mathcal{S}_1$, a corresponding item in $\mathcal{S}_2$ can be found in $O(1)$ time. The entry for the item can be inserted in $\mathcal{S}_C$ in $O(1)$ time and, if the item has been found in the other summary, the corresponding entry can be deleted from $\mathcal{S}_2$ again in $O(1)$ time. Since there are at most k entries in a summary, scanning and processing the first summary requires $O(k)$.

Next, the entries in $\mathcal{S}_2$ are scanned (note that there can be at most k entries in $\mathcal{S}_2$: this may happen only when the items in the two summaries are all distinct, otherwise there will be less than k entries because corresponding items are removed from $\mathcal{S}_2$ each time a match is found). For each entry in $\mathcal{S}_2$, the corresponding item is inserted in $\mathcal{S}_C$ in $O(1)$ time. Therefore, processing $\mathcal{S}_2$ requires in the worst case $O(k)$ time.

The combined summary $\mathcal{S}_C$ is returned as is if its total number of entries is less than or equal to k, otherwise only the last k entries (i.e., those entries corresponding to the items with greatest frequencies) are returned. The time required is $O(k)$.

Therefore, at most $O(k)$ work is done in each step of the parallel reduction, and there are $O(\log\ p)$ such steps. It follows that a parallel reduction can be done in $O(k\log p)$ time in the worst case.

The parallel complexity of our algorithm is therefore

$$T_p = O(n/p + (k_1 + k_2) \log p) \tag{7}$$

in the worst case. Finally, the complexity of a query (Algorithm 4) is simply $O(k_1 + k_2)$, owing to the fact that we simply need to perform a linear scan of both summaries, and the work done processing each entry is $O(1)$ in the worst case.

We now analyze, from a theoretical perspective, our algorithm. Since the sequential algorithm requires $T_1 = O(n)$ time in the worst case [17], the parallel overhead is $T_o = pT_p - T_1 = p(k_1 + k_2) \log p$. The isoefficiency is then given by

$$n \geq Cp(k_1 + k_2) \log p \tag{8}$$

where C is a constant. If we consider k_1 and k_2 to be constants as well, then the isoefficiency function is given by $p \log p$. Even though the algorithm is not perfectly scalable, it is only a small factor ($\log p$) away from optimality.

5 Experimental Results

In order to evaluate the parallel algorithm for mining CHHs, we have implemented PCSSCHH in C++. The source code has been compiled using the Intel C++ compiler v15.0.3 and the Intel MPI library v5.0.3 on Linux CentOS distribution with the following flags: -O3 -xHost -std=c++11. The tests have been carried out on "Athena" parallel cluster kindly provided by the Euro-Mediterranean Center on Climate Changes, Foundation (CMCC) in Italy. The cluster is made of 482 computational nodes, each one equipped with 64 GB of RAM and two Intel 2.60 GHz octa-core Xeon E5-2670 processors. The source code is freely available for inspection and for reproducibility of results contacting the authors by email.

The synthetic datasets used in our experiments are distributed according to the Zipf distribution. In each one of the experiments, the execution has been repeated 5 times using a different seed for the pseudo-random number generator used for creating the input data stream. For each input distribution generated, the results have been averaged over all of the runs. The input items are tuples of two 32 bits unsigned integers.

The experiments are aimed at evaluating the parallel algorithm behavior in terms of performance and accuracy. We used the parallel speedup, efficiency and scaled speedup metrics to measure the computational performance and the *precision* and *recall* to evaluate the impact of the parallelization on the algorithm's accuracy. In the experiments, we vary the following input parameters: length of the data stream (ni), skewness of the zipfian distribution (ρ), number of counters (k_1, k_2) used in the primary and secondary summaries and the number of processes (p). For the scaled speedup measurement we used three different values of *grain size*, i.e., the number of tuples assigned to a process; in the scaled speedup analysis (also known as weak scalability) the grain size is kept constant while the number of processes increases.

Table 1. Experiments carried out.

Parameter	Values	Default
Stream size (ni, billions)	1, 3, 6	1
Skew (ρ)	1.0, 1.4, 1.8	1.4
Counters (k_1, k_2)	$k_1 = 833, k_2 = 2633$	$k_1 = 3050, k_2 = 12450$
	$k_1 = 1033, k_2 = 3500$	
	$k_1 = 1673, k_2 = 6323$	
	$k_1 = 3050, k_2 = 12450$	
Grain Size (ni, millions)	100, 500, 750	-
Num. Procs. (p)	1, 16, 32, 64, 128, 256, 512	-

Table 1 reports all of the parameters used in our experiments.

We begin our analysis discussing the computational performance of the algorithm. Figures 1 and 2 show the parallel speedup and efficiency measured varying respectively the stream size and the skew of the input distribution, while Fig. 3 shows the corresponding elapsed times (the plots of Figs. 1a, 2a, 3a and b use a log-log scale, whilst Figs. 1b and 2b use a log scale on the x axis, related to the number of processes).

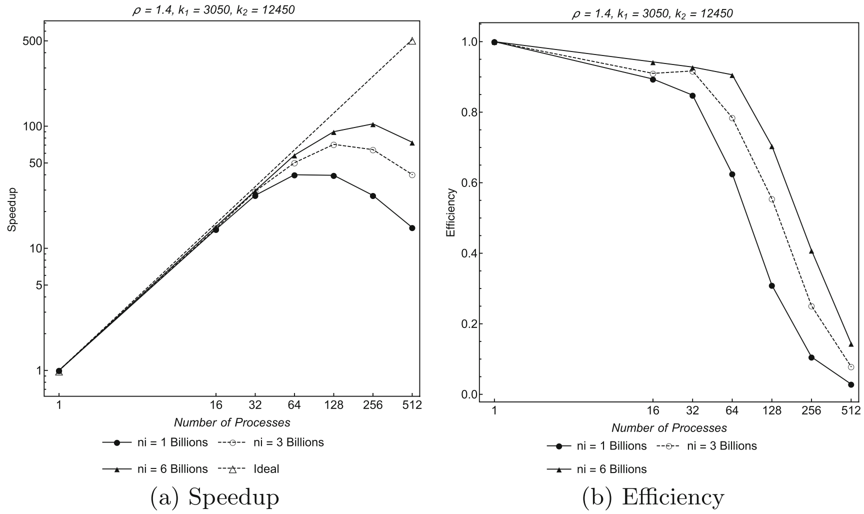
According to the Amdahl's law, when the problem size increases, the parallel algorithm performs better (the so-called Amdahl's effect), and shows very good scalability up to 64 processes. In order to use a greater number of processes, owing to the isoefficiency analysis reported in the previous Section, we need to increase the problem size.

For low skew values, the time required to update a summary increases owing to the higher number of cache misses due to a greater number of distinct items, which also increases the hash table's access time. Since the update time increases but the communication time remains the same, the speedup improves.

In Fig. 4 the scaled speedup is reported, using a log-log scale. The results clearly show that the algorithm performs well when the grain size is increased.

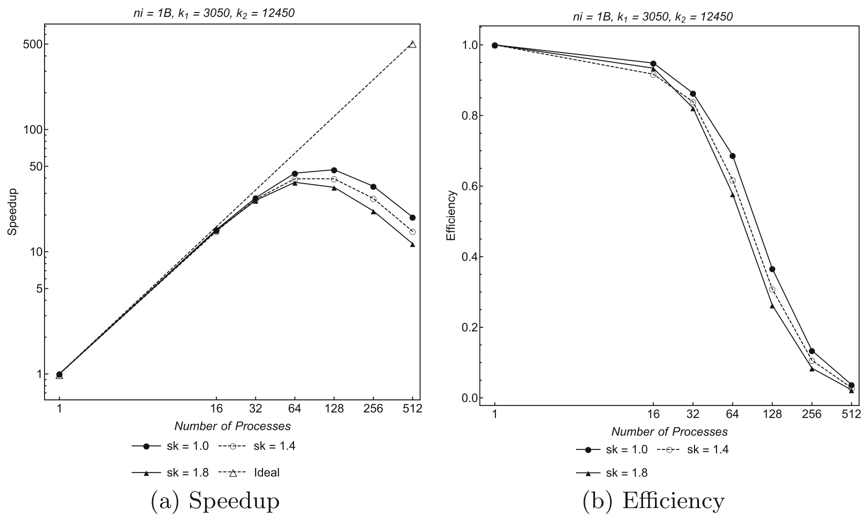
Regarding the accuracy of the algorithm, we measured *recall* and *precision*. The recall is the total number of true frequent items reported over the number of frequent items given by an exact algorithm. Therefore, an algorithm is correct iff its recall is equal to one. Since in all of the experiments we used a number of counters k_1, k_2 that guarantees the correctness of the algorithm, we observed a recall equal to one in each experiment. For this reason, the corresponding plots have not been reported.

Since precision is defined as the total number of true frequent items reported over the total number of items reported, this metric quantifies the number of false positives outputted by an algorithm. The accuracy of the sequential algorithm has been already well analyzed in [17]; here, the goal is to prove that the parallelization does not impact on the accuracy. Figure 5 reports, using a log scale for the x axis, the precision obtained varying the number of counters. As



(a) Speedup (b) Efficiency

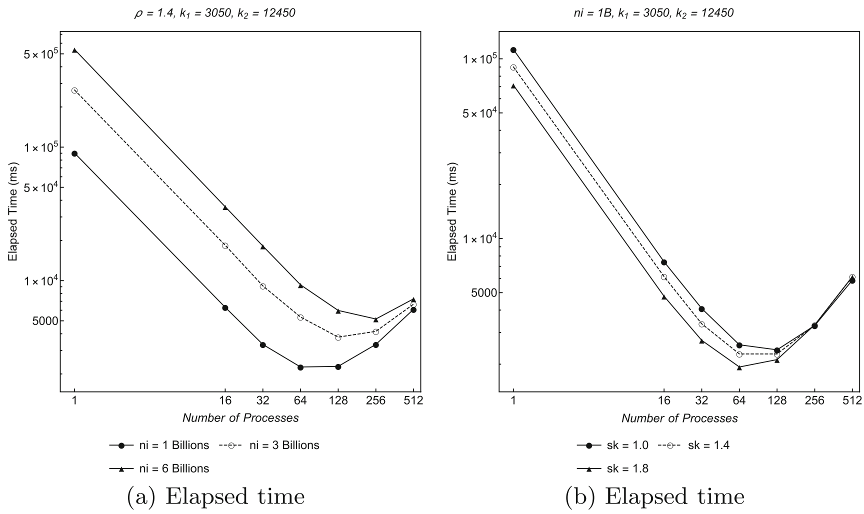
Fig. 1. Speedup and efficiency varying the input stream size.



(a) Speedup (b) Efficiency

Fig. 2. Speedup and efficiency varying the skew of the distribution.

expected, when we use a few counters the algorithm is less accurate. The results also show that the parallelization, and in particular the reduction in which we merge the summaries, does not introduce any significant estimation error, indeed the precision does not change varying the number of processes.



(a) Elapsed time (b) Elapsed time

Fig. 3. Elapsed time varying the stream size and the skew of the distribution.

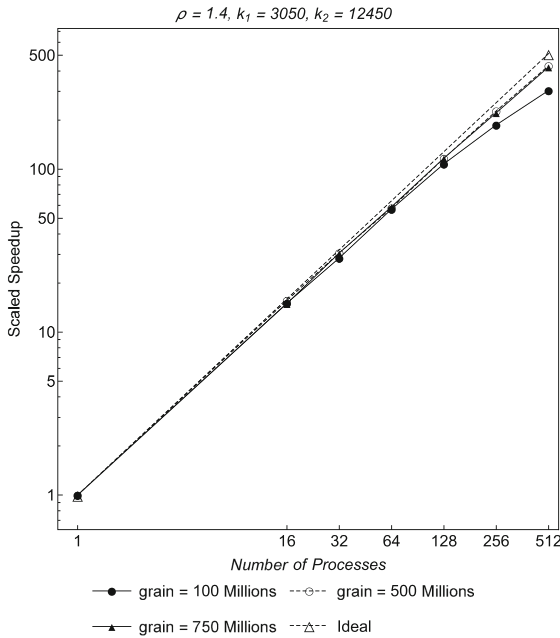


Fig. 4. Scaled speedup varying the grain size.

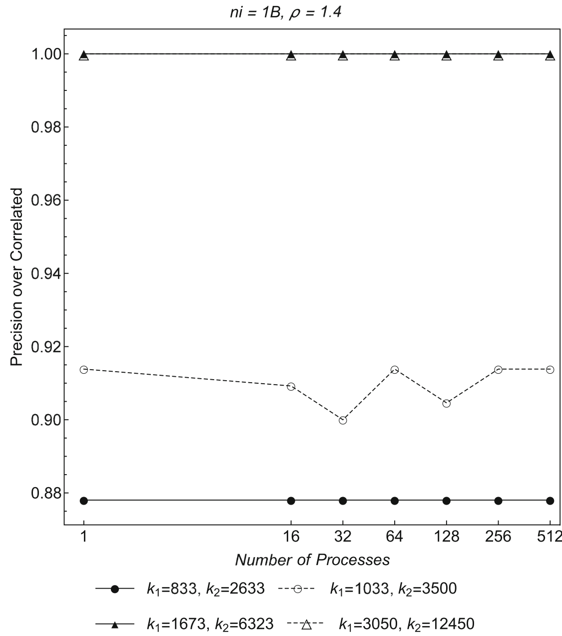


Fig. 5. Precision varying the number of processes and the number of counters in the primary and secondary stream summaries.

6 Conclusions

In this paper we have presented a message-passing based parallel algorithm for mining Correlated Heavy Hitters from a two-dimensional data stream. We have shown, through extensive experimental results, that our algorithm solves the ACHH problem and provides very good scalability, whilst retaining the accuracy of its sequential counterpart. To the best of our knowledge, this is the first parallel algorithm solving the problem.

Acknowledgements. The authors would like to thank the Supercomputing Center of the Euro-Mediterranean Center on Climate Changes, Foundation for granting the access to the Athena supercomputer machine.

References

1. Boyer, R., Moore, J.: MJRTY - a fast majority vote algorithm. Technical report 32, Institute for Computing Science, University of Texas, Austin (1981)
2. Boyer, R., Moore, J.S.: MJRTY - a fast majority vote algorithm. In: Boyer, R.S. (ed.) Automated Reasoning: Essays in Honor of Woody Bledsoe. Automated Reasoning Series, pp. 105–117. Kluwer Academic Publishers, Dordrecht (1991)

3. Cafaro, M., Epicoco, I., Aloisio, G., Pulimeno, M.: CUDA based parallel implementations of space-saving on a GPU. In: 2017 International Conference on High Performance Computing Simulation (HPCS), pp. 707–714, July 2017. https://doi.org/10.1109/HPCS.2017.108
4. Cafaro, M., Epicoco, I., Pulimeno, M., Aloisio, G.: On frequency estimation and detection of frequent items in time faded streams. IEEE Access **5**, 24078–24093 (2017). https://doi.org/10.1109/ACCESS.2017.2757238
5. Cafaro, M., Pulimeno, M.: Merging frequent summaries. In: Proceedings of the 17th Italian Conference on Theoretical Computer Science (ICTCS 2016), vol. 1720. pp. 280–285. CEUR Proceedings (2016)
6. Cafaro, M., Pulimeno, M., Epicoco, I.: Parallel mining of time-faded heavy hitters. Expert Syst. Appl. **96**, 115–128 (2018). https://doi.org/10.1016/j.eswa.2017.11.021, http://www.sciencedirect.com/science/article/pii/S0957417417307777
7. Cafaro, M., Pulimeno, M., Epicoco, I., Aloisio, G.: Mining frequent items in the time fading model. Inf. Sci. **370–371**, 221–238 (2016). https://doi.org/10.1016/j.ins.2016.07.077
8. Cafaro, M., Pulimeno, M., Epicoco, I., Aloisio, G.: Parallel space saving on multi- and many-core processors. Concurr. Comput.: Pract. Exp. **30**(7), e4160-n/a (2017). https://doi.org/10.1002/cpe.4160
9. Cafaro, M., Pulimeno, M., Tempesta, P.: A parallel space saving algorithm for frequent items and the hurwitz zeta distribution. Inf. Sci. **329**, 1–19 (2016). https://doi.org/10.1016/j.ins.2015.09.003, http://www.sciencedirect.com/science/article/pii/S002002551500657X
10. Cafaro, M., Tempesta, P.: Finding frequent items in parallel. Concurr. Comput.: Pract. Exp. **23**(15), 1774–1788 (2011). https://doi.org/10.1002/cpe.1761
11. Charikar, M., Chen, K., Farach-Colton, M.: Finding frequent items in data streams. In: Widmayer, P., et al. (eds.) ICALP 2002. LNCS, vol. 2380, pp. 693–703. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45465-9_59
12. Chen, L., Mei, Q.: Mining frequent items in data stream using time fading model. Inf. Sci. **257**, 54–69 (2014). https://doi.org/10.1016/j.ins.2013.09.007, http://www.sciencedirect.com/science/article/pii/S0020025513006403
13. Cormode, G., Muthukrishnan, S.: An improved data stream summary: the count-min sketch and its applications. J. Algorithms **55**(1), 58–75 (2005). https://doi.org/10.1016/j.jalgor.2003.12.001
14. Cormode, G., Muthukrishnan, S.: What's hot and what's not: tracking most frequent items dynamically. ACM Trans. Database Syst. **30**(1), 249–278 (2005). https://doi.org/10.1145/1061318.1061325
15. Das, S., Antony, S., Agrawal, D., El Abbadi, A.: Thread cooperation in multicore architectures for frequency counting over multiple data streams. Proc. VLDB Endow. **2**(1), 217–228 (2009). https://doi.org/10.14778/1687627.1687653
16. Demaine, E.D., López-Ortiz, A., Munro, J.I.: Frequency estimation of internet packet streams with limited space. In: Möhring, R., Raman, R. (eds.) ESA 2002. LNCS, vol. 2461, pp. 348–360. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45749-6_33
17. Epicoco, I., Cafaro, M., Pulimeno, M.: Fast and accurate mining of correlated heavy hitters. Data Min. Knowl. Discov. **32**(1), 162–186 (2018). https://doi.org/10.1007/s10618-017-0526-x

18. Erra, U., Frola, B.: Frequent items mining acceleration exploiting fast parallel sorting on the GPU. Proc. Comput. Sci. **9**, 86–95 (2012). https://doi.org/10.1016/j.procs.2012.04.010, http://www.sciencedirect.com/science/article/pii/S1877050912001317. Proceedings of the International Conference on Computational Science, ICCS 2012
19. Govindaraju, N.K., Raghuvanshi, N., Manocha, D.: Fast and approximate stream mining of quantiles and frequencies using graphics processors. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, SIGMOD 2005, pp. 611–622. ACM (2005). https://doi.org/10.1145/1066157.1066227
20. Jin, C., Qian, W., Sha, C., Yu, J.X., Zhou, A.: Dynamically maintaining frequent items over a data stream. In: Proceedings Of CIKM, pp. 287–294. ACM Press (2003)
21. Karp, R.M., Shenker, S., Papadimitriou, C.H.: A simple algorithm for finding frequent elements in streams and bags. ACM Trans. Database Syst. **28**(1), 51–55 (2003). https://doi.org/10.1145/762471.762473
22. Lahiri, B., Mukherjee, A.P., Tirthapura, S.: Identifying correlated heavy-hitters in a two-dimensional data stream. Data Min. Knowl. Disc. **30**(4), 797–818 (2016). https://doi.org/10.1007/s10618-015-0438-6
23. Manku, G.S., Motwani, R.: Approximate frequency counts over data streams. In: VLDB, pp. 346–357 (2002)
24. Metwally, A., Agrawal, D., Abbadi, A.E.: An integrated efficient solution for computing frequent and top-k elements in data streams. ACM Trans. Database Syst. **31**(3), 1095–1133 (2006). https://doi.org/10.1145/1166074.1166084
25. Misra, J., Gries, D.: Finding repeated elements. Sci. Comput. Program. **2**(2), 143–152 (1982)
26. Roy, P., Teubner, J., Alonso, G.: Efficient frequent item counting in multi-core hardware. In: Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2012, pp. 1451–1459. ACM (2012). https://doi.org/10.1145/2339530.2339757
27. Tangwongsan, K., Tirthapura, S., Wu, K.L.: Parallel streaming frequency-based aggregates. In: Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2014, pp. 236–245. ACM (2014). https://doi.org/10.1145/2612669.2612695
28. Wu, S., Lin, H., Gao, Y., Lu, D.: Novel structures for counting frequent items in time decayed streams. World Wide Web **20**(5), 1111–1133 (2017). https://doi.org/10.1007/s11280-017-0433-5
29. Zhang, Y.: Parallelizing the weighted lossy counting algorithm in high-speed network monitoring. In: Second International Conference on Instrumentation, Measurement, Computer, Communication and Control (IMCCC), pp. 757–761 (2012). https://doi.org/10.1109/IMCCC.2012.183
30. Zhang, Y., Sun, Y., Zhang, J., Xu, J., Wu, Y.: An efficient framework for parallel and continuous frequent item monitoring. Concurr. Comput.: Pract. Exp. **26**(18), 2856–2879 (2014). https://doi.org/10.1002/cpe.3182