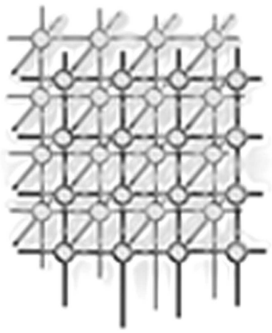
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2007; **00**:1–8 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

Finding Frequent Items in Parallel



Massimo Cafaro[1*,†], Piergiulio Tempesta[2]

[1] *University of Salento, Lecce, Italy,*
CMCC - Euro-Mediterranean Centre for Climate Change, Lecce, Italy
[2] *Universidad Complutense, Madrid, Spain*

SUMMARY

We present a deterministic parallel algorithm for the k–majority problem, that can be used to find in parallel frequent items, i.e., those whose multiplicity is greater than a given threshold, and is therefore useful in the context of iceberg queries and many other different contexts. The algorithm can be used both in the on–line (stream) context and in the off–line setting, the difference being that in the former case we are restricted to a single scan of the input elements, so that verifying the frequent items that have been determined is not allowed (e.g., network traffic streams passing through internet routers), while in the latter a parallel scan of the input can be used to determine the actual k–majority elements. To the best of our knowledge, this is the first parallel algorithm solving the proposed problem.

KEY WORDS: Data Stream, Frequent Elements

1. INTRODUCTION

In order to state the k–majority problem, let us recall some basic definitions related to multiset theory. [1].

Definition 1.1. A multiset is a pair (A, m), where A is some set, called the underlying set of elements, and $m : A \to \mathbb{N}$ is a function.

Definition 1.2. Let $A \subset X$ be a multiset. The indicator function of A is

$$I_A(x) = \begin{cases} m(x) & x \in A \\ 0 & x \notin A. \end{cases} \quad (1)$$

This definition generalizes that of indicator function in standard set theory, as a function taking values in $\{0, 1\}$.

*Correspondence to: Facoltà di Ingegneria, Università del Salento, Via per Monteroni, 73100 Lecce Italy
†E-mail: massimo.cafaro@unisalento.it

Corollary. *The cardinality of A is expressed by*

$$Card(A) = \sum_{x \in X} I_A(x). \tag{2}$$

In the sequel, the set A will play the role of a finite input array, containing n elements. The integer–valued function m, for each $x \in A$, will provide the *multiplicity* or number of occurrences of x in A, i.e. $m_x^A \equiv m(x) = Card(\{i : A[i] = x\})$.

Our problem can now be stated formally as follows.

Definition 1.3. Given a *multiset* A of n elements, a k–majority element is an element $x \in A$ whose *multiplicity* $m(x)$ is such that $m(x) \geq \lfloor \frac{n}{k} \rfloor + 1$.

Definition 1.4. *k–Majority problem.*
Input: An array A of n numbers.
Output: The set $M = \left\{x : m(x) \geq \lfloor \frac{n}{k} \rfloor + 1\right\}$.

For $k = 2$, the problem reduces to the well known majority problem [2, 3] [4]. Finding frequent items, those whose multiplicity is greater than a given threshold, is referred to in the literature as an *iceberg query* [5] [6] owing to the fact that the number of k–majority elements is often very small (the tip of an iceberg), relative to the large amount of input data (the iceberg). Another name for this kind of queries is *hot list analysis* [7].

Besides being important from a theoretical perspective, algorithms for this problem are also extremely useful in practical contexts such as, for instance, in all the cases (such as electronic voting) where a quorum of more than n/k of all the votes received is required for a candidate to win; other good examples are related to the extraction of essential characteristics of network traffic streams passing through internet routers such as frequency estimation of internet packet streams [8] and monitoring internet packet to infer network congestion [9] [10]. Additional applications include market basket analysis in data mining [11] and analysis of web query logs [12]. We note here that the class of applications considered here is characterized by the condition $k = O(1)$.

The problem is also relevant in Computational Linguistics, for instance in connection with the estimation of the frequencies of specific words in a given language [13], or in all contexts where a verification of the Zipf–Mandelbrot law is required [14] [15](theoretical linguistics, ecological field studies [16], etc.).

The k–majority problem has been solved sequentially first by Misra and Gries [17]. In their paper, it is shown how to solve it in time $O(n \log k)$. The basic idea is that given a *multiset*, i.e., a set of elements with duplicates allowed (also called a *bag*), 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 data structure used in the implementation of their algorithm is an AVL tree [18].

An optimal algorithm by Demaine et al., with worst-case complexity of $O(n)$, was published recently [8]. This deterministic algorithm called *Frequent* is, in practice, identical to the one developed by Misra and Gries; however, through a better use of data structures (a doubly linked list of groups to support decrementing a set of counters at once in constant time) it achieves a complexity of $O(n)$. Another algorithm was discovered independently by Karp et

al. and published almost simultaneously [19]. This algorithm is, in turn, practically equivalent to the one by Demaine et al. It is based on hashing operations and achieves the $O(n)$ bound on average. Since then, many other algorithms were developed to improve space requirements etc.

Cormode and Hadjieleftheriou present in a very recent work [20] a survey of existing algorithms, classifying them as either counter–based or sketch–based. The *Frequent* algorithm belongs to the former class; *LossyCounting* [21] and *SpaceSaving* [22] are other notable examples of counters–based algorithms. Among the sketch–based ones, we recall here *CountSketch* [12] and *CountMin* [23]. In the concluding remarks, Cormode and Hadjieleftheriou note that "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 this paper, we deal with the strictly related parallel case and show how to merge in parallel multiple counter–based summaries. We present a deterministic parallel algorithm for the k–majority problem, in the context of message–passing architectures. The algorithm can be used both in the on–line (stream) context and in the off–line setting, the difference being that in the former case we are restricted to a single scan of the input elements, so that verifying the frequent items that have been determined is not allowed (e.g., network traffic streams passing through internet routers), while in the latter a parallel scan of the input can be used to determine the actual k–majority elements. To the best of our knowledge, this is the first parallel algorithm solving the problem.

This article is organized as follows. Our 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.

2. THE PARALLEL ALGORITHM

Given an input array A with n elements, let p be the number of processors we use in parallel. The pseudocode of Algorithm 2.1 describes our parallel majority algorithm. The initial call is $ParallelFrequentItems(A, n, p)$.

The algorithm determines potential k–majority elements; it is worth recalling here that, when dealing with streams, scanning the input elements again to verify if the candidates found are actual k-majority elements is not permitted. However, in the off–line setting, the Algorithm 2.3 can be used to verify in parallel the candidate elements.

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 its local candidates and their corresponding weights in step 3, by utilizing the well–known algorithm designed by Demaine et al. [8], shown in the

pseudocode as the *Frequent* function. The weights are the values associated to the counters greater than zero corresponding to the k–majority candidates determined.

The pair $(local, weights)$ 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 *ParallelCandidateReductionOperator* function, shown in the text as Algorithm 2.2.

At each step of the reduction, a processor receives as input from two other processes a couple of multisets (arrays) containing respectively candidate elements and their weights. The Algorithm 2.2 creates at the beginning two arrays X and Y containing respectively candidate elements and their weights. This is done in steps 1–8. Then, a slightly modified version of the *Frequent* algorithm is used to determine a set of candidates and their weights which are then passed as input to another processor to continue the parallel reduction.

Our modified version of the *Frequent* algorithm is based on the fact that, for a given candidate element in the array X, the corresponding entry in the array Y provides its weight, which is used as the number of occurrences of the candidate element. We initialize the *weights* and *candidates* arrays respectively in steps 9 and 10. The *candidates* array will be returned as the output of the algorithm. Step 11 determines the length of both X and Y.

The for loop of step 12 scans the elements. We repeat the following operations until the weight of the current element being examined is greater than zero in the *while* loop of step 13. If the element being examined is not monitored by any counter and there is a counter available to monitor it (i.e., a counter whose value is zero), then we assign the current element to the available counter and update the *candidate* array which keeps track of the elements assigned to the counters. This is done in steps 14–16.

If the current element being examined has been assigned to a counter c, then in steps 17–20 we increment this counter by the weight of the same element, reset the weight of the current element to zero and keep track of the current minimum among the counters which are monitoring elements. While in the original algorithm the occurrences of a particular element are found one at a time, in our algorithm we know in advance that an element may occur at most two times, one in the $local_i$ and one in the $local_j$ input arrays. Therefore, each time we process an element, we can increment one–shot the counter in charge of monitoring it by the corresponding number of occurrences, given by its weight. The minimum among the counters is used when no counter is available.

In this case, all the $k-1$ counters have been assigned to corresponding $k-1$ candidate elements. Therefore the current element being examined is the kth one. As in the original algorithm, we need to decrement all the counters at once. However, in our case the decrement depends on both the weight of the current element and the minimum among the counters we have already determined. Owing to the fact that we can not let a counter's value go negative, in steps 21–26 we determine m, the minimum between the weight of the element being considered and cm, the minimum among the counters. Then we decrement all the counters and the weight of our current element by m. If a counter's value reaches zero after the decrement, we also update correspondingly the *candidates* array. Finally, the algorithm returns in step 29 both the *candidates* and *weights* arrays.

Assuming that at the end of the parallel reduction the processor whose rank is zero holds the result of the parallel reduction, steps 5–6 carried out by the processor with rank zero either return the set of potential k–majority elements or the empty set. If the verification of the

candidates is allowed, i.e. we are dealing with the off–line setting, the Algorithm 2.3 can be used to verify the candidates in parallel. Each processor determines the number of occurrences of the global candidates in its local sub–array in step 3. Finally, the processors engage in a parallel sum reduction in step 4 to determine *counts*, an array containing the number of occurrences of the global candidates for the whole array A.

Again, assuming that at the end of this parallel sum reduction the processor whose rank is zero holds the result, steps 5–11 carried out by the processor with rank zero either output a set containing the k–majority elements of A or the empty set. We initialize $frequent$ in step 6 to an empty set; this is the set that will be returned as the final output of our algorithm. Steps 7–10 update $frequent$ depending on the test $counts[e] > n/k$.

Algorithm 2.1: ParallelFrequentItems

Input: A, an array; n, the length of A; p, the number of processors
Output: an array containing k–majority candidate elements

```
/* The n elements of the input array A are distributed to the p processors
   so that each one is responsible for either ⌊n/p⌋ or ⌈n/p⌉ 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 ← ⌊id n/p⌋;
2 right ← ⌊(id + 1) n/p⌋ − 1;
  /* determine local candidates */
3 local, weights ← Frequent(A, left, right);
  /* determine the global candidates for the whole array */
4 global ← ParallelCandidateReduction(local, weights);
  /* we assume here that the processor with rank 0 holds the final result of
     the parallel reduction */
5 if id == 0 then
6     return global;
```

3. PROOF OF ALGORITHM'S CORRECTNESS

The correctness of the algorithm rests on the following Lemmas and Theorem.

Let $c_j^{A_i}$, $0 \leq j \leq k-2$ be the local k–majority candidates in the subset (sub–array) A_i, consisting of e_i elements, analyzed by the processor p_i, whose rank, denoted by id, satisfies the inequalities $0 \leq id \leq p-1$. Let $m_j^{A_i} \equiv m(c_j^{A_i})$ denote the multiplicity of $c_j^{A_i} \in A_i$. Let also γ_i denote the cardinality (or length of the sub–array) of A_i and $n = \sum_i \gamma_i$ the cardinality of $A = \biguplus_i A_i$. Here the $\uplus$ operator denotes the *join operation* [1], which is the sum of the multiplicity functions as follows: $I_{A \uplus B}(x) = I_A(x) + I_B(x)$.

From the block distribution used for the initial domain decomposition, it follows that

$$e_i = \lfloor (id+1)\ n/p \rfloor - \lfloor id\ n/p \rfloor . \tag{3}$$

Algorithm 2.2: ParallelCandidateReductionOperator

Input: $local_i$ and $local_j$, arrays of local candidate elements for processors p_i and p_j; $weights_i$ and $weights_j$, arrays containing the weights of the candidates in $local_i$, $local_j$; n_i and n_j, the lengths of the previous arrays

Output: arrays containing k–majority candidate elements and their weights

`/* create the array` X `containing the elements in` $local_i$`,` $local_j$ `and the array` Y `containing the weights in` $weights_i$`,` $weights_j$ `*/`

for $z = 0$ **to** n_i*-1* **do**
    put $local_i[z]$ in X;
    put $weights_i[z]$ in Y ;
end
for $z = 0$ **to** n_j*-1* **do**
    put $local_j[z]$ in X;
    put $weights_j[z]$ in Y;
end

`/* the` *weights* `array will contain` $k-1$ `counters */`

Initialize the *weights* array to zero;

`/* the` *candidates* `array will contain the elements associated to the counters */`

initialize the *candidates* array;
$l = n_i + n_j$;
for $z = 0$ **to** $l - 1$ **do**
    while $Y[z] > 0$ **do**
        if $X[z]$ *is not monitored by any counter and some counter is zero* **then**
            let $X[z]$ be the monitored element of that counter;
            update as needed the *candidates* array;
        if $X[z]$ *is the monitored element of a counter* c **then**
            $c = c + Y[z]$;
            $Y[z] = 0$;
            cm = minimum among the counters' values;
        else
            m = minimum between $Y[z]$ and cm;
            decrement every counter by m;
            if *a counter is zero* **then**
                update as needed the *candidates* array;
            $Y[z] = Y[z] - m$;
    end
end
return *candidates*, *weights*;

Algorithm 2.3: ParallelVerify

Input: A, an array; n, the length of A; p, the number of processors; *candidates*, an array containing the k–majority candidates

Output: a set containing the k–majority elements of A

```
/* The n elements of the input array A are distributed to the p processors
   so that each one is responsible for either ⌊n/p⌋ or ⌈n/p⌉ 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 */
left ← ⌊id n/p⌋;
right ← ⌊(id + 1) n/p⌋ − 1;
/* determine the occurrences of the global candidates */
multiplicities ← Occurrences(A, candidates, left, right);
/* determine the occurrences for the whole array */
counts ← ParallelSumReduction(multiplicities);
/* we assume here that the processor with rank 0 holds the final result of
   the parallel reduction */
if id == 0 then
    frequent = ∅;
    foreach e in candidates do
        if counts[e] > n/k then
            put e in frequent;
    end
    return frequent;
```

Lemma 3.1. *Let $x \in A$ be an element such that $m_x^{A_i} \leq n/(p\ k)$, $\forall i \in \{0, \ldots, p-1\}$. Then, the element x can not be a k–majority element.*

Proof. Indeed,

$$\sum_{i-0}^{p-1} m_x^{A_i} \leqslant \frac{n}{k} < \frac{n}{k} + 1. \tag{4}$$

□

Lemma 3.2. *Given an array 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 k distinct k–majority elements. It follows that the array A must contain at least $k\ (\lfloor n/k \rfloor + 1)$ elements. Since $\lfloor n/k \rfloor + 1 > n/k$, we deduce that $k\ (\lfloor n/k \rfloor + 1) > k\ (n/k) = n$, thus contradicting the hypothesis that $Card(A) = n$. □

Definition 3.1. The *weight* $w^r(c_i)$ of a candidate element c_i is the number of its occurrences after that the parallel reduction process has been performed r times.

In what follows, we will omit the superscript r if it is not necessary to specify the iteration we are considering.

Following the notation and terminology in [8], given an array A, suppose that a candidate c_i is read t_f times when other counters are full, and t_i times when the candidate is already assigned a counter. Therefore, the value of the counter associated to c_i is $t = t_f + t_i$. Finally, let t_d denote the number of times that the counter assigned to c_i is decremented as another value is read. We deduce that

$$w(c_i) = t_i - t_d.$$

Since a counter never goes negative, $w(c_i) \geq 0$. In [8], the following inequality has been proved:

$$w(c_i) > 0. \tag{5}$$

This inequality is sufficient to prove the correctness of the sequential algorithm *Frequent*. However, we propose here a refinement of the simple estimation given in equation (5), that might be useful.

Theorem 3.3. *For any sub–array $A_q (q = 0, ..., p-1)$, of cardinality γ_q, given a k–majority element $c_i \in A_q$, we have the following estimation:*

$$\min w(c_i) \geq O(c_i) - \left\lfloor \frac{\gamma_q - (O(c_i) + k - 2)}{k-1} \right\rfloor. \tag{6}$$

Proof. It is easy to ascertain that, in a given sub–array A_q, a counter associated to a given k–majority element c_i is decremented maximally if in the data stream all the remaining elements are distinct. Let us prove first this case, that corresponds to the equality in (6). Assume, for instance, that the element fills a counter, with occurrences $O(c_i)$. The other $k-2$ counters will also be filled by the first $k-2$ elements read from the stream. The first of the subsequent elements will decrease by one all the counters, so that $k-2$ of them go to zero, and will be filled by other $k-2$ distinct elements. This process will be repeated a number of times exactly equal to the number φ of $(k-1)$–ples we can construct from the elements distinct from the element, i.e.

$$\varphi = \left\lfloor \frac{\gamma_q - (O(c_i) + k - 2)}{k-1} \right\rfloor.$$

This proves the equality in (6), in the considered case. The inequality follows a fortiori in all the other cases. □

In particular, let us consider the particular case when we have a unique array A and $O(c_i) = \lfloor \frac{n}{k} \rfloor + 1$. This corresponds to the minimal number of occurrences necessary in order to ensure the existence of a k–majority element.

The previous Theorem 3.3 provides the interesting estimation:

$$\min w(c_i) \geq \left\lfloor \frac{n}{k} \right\rfloor + 1 - \left\lfloor \frac{n - \lfloor \frac{n}{k} \rfloor}{k-1} - 1 \right\rfloor. \tag{7}$$

This formula contains the proof of [8], due to the next result.

Lemma 3.4. *For any $n, k \in \mathbb{N}$, with $k \geq 2$, the following inequality holds*

$$\left\lfloor \frac{n}{k} \right\rfloor + 1 - \left\lfloor \frac{n - \left\lfloor \frac{n}{k} \right\rfloor}{k-1} - 1 \right\rfloor > 0. \tag{8}$$

Proof. Put $\left\lfloor \frac{n}{k} \right\rfloor = \frac{n}{k} + d$, with $0 < d < 1$. We get

$$\min w(c_i) \geq \frac{n}{k} + d + 1 - \left\lfloor \frac{n - \frac{n}{k} - d}{k-1} - 1 \right\rfloor =$$

$$\frac{n}{k} + d + 1 - \left\lfloor \frac{n}{k} - \frac{d}{k-1} - 1 \right\rfloor,$$

which is strictly greater than zero for $k \geq 2$. □

Consequently, we have proven that

$$\min w(c_i) \geq 1, \tag{9}$$

which is equivalent to the statement of correctness of the algorithm *Frequent* of [8].

Theorem 3.5. *Algorithm 2.1 correctly determines k–majority candidate elements.*

Proof. Assume now that at least one majority element exist. Once the initial data stream has been distributed to all p processors, and given a specific sub–array A_i, the algorithm assigns to it $k-1$ counters $a_1, \ldots a_{k-1}$. The value of a counter a_i is incremented by one if the element being considered is equal to the candidate element associated to the counter, otherwise all the counters are decremented by one. This is clearly equivalent to deleting k distinct elements, whenever is possible.

If an element c_i is of k–majority, after deleting k distinct elements, it is still a k–majority element in the reduced block it belongs to. Indeed, to prove this simple statement, consider the case when the occurrences of the k–majority element in a block A_q of cardinality γ_q are minimal, i.e. $O(c_i) = \left\lfloor \frac{\gamma_q}{k} \right\rfloor + 1$. After deleting k distinct elements, the occurrences of c_i are decremented by one, i.e. $O(c_i) = \left\lfloor \frac{\gamma_q}{k} \right\rfloor = \left\lfloor \frac{\gamma_q - k}{k} \right\rfloor + 1$, namely c_i is of k–majority in the reduced block, containing now $\gamma_q - k$ elements. The same property holds when the occurrences $O(c_i)$ are not minimal.

Now consider the rth step of the parallel reduction; this amounts to merging into a new multiset A^r different reduced multisets $A_0^r, A_1^r, \ldots$, each of them provided by the parallel reduction process, such that $\biguplus_i^w A_i^r = A^r$. Here $A^0 \equiv A$ and $w = \frac{p}{2^r}$.

The occurrences of the elements being considered in the rth step are equal to the weights $w^r(c_i)$ provided by the sequential algorithm. The *ParallelCandidateReductionOperator* combines intermediate results, and still removes k–ples of distinct elements across the new sub–arrays generated by the parallel reduction.

Observe that Lemma 3.1 is equivalent to saying that there exists at least one sub–array in which the element c_i is of k–majority, if it is *globally* (namely for the initial input stream of data) a k–majority element. Therefore, after the first step of the parallel reduction, c_i is again

of k–majority in at least one of the sub–arrays resulting by merging pairwise the reduced bags corresponding to each processor. Iterating the steps of the parallel reduction, we arrive at a final reduced bag. Since Lemma 3.1 still holds, the candidate c_i must necessarily appear in the final bag, i.e. its related counter does not go to zero. This completes the proof. □

There are several similar ways to prove the Theorem 3.5. This one seems to us the quickest one.

4. ANALYSIS

In this Section, we derive the parallel complexity of Algorithm 2.1. At the beginning of the $ParallelFrequentItems$ algorithm, the workload is balanced by using a block distribution; this is done in steps 1–2 with two simple assignments. Therefore, the complexity of the initial domain decomposition is $O(1)$. Determining local candidates in step 3 by using the *Frequent* algorithm requires $O(n/p)$ time since the complexity of this algorithm is linear in the number of its input elements.

The $ParallelCandidateReductionOperator$ function, shown in the text as Algorithm 2.2, is used internally in the parallel reduction step. The complexity of this algorithm can be derived as follows. A processor receives as input from two processors their arrays of candidate elements and the corresponding arrays of weights. By Lemma 3.2, the size of each array of candidate elements is at most $k-1$. It follows that the size of the arrays X and Y constructed in steps 1–8 is at most $2(k-1) = O(k)$ and this is also the time required to build the arrays.

Initializing the candidates and weights array requires $O(k)$; it can even be done in $O(1)$ time by using the data structure described in [24]. However, in our case a worst case complexity of $O(k)$ for this initialization step does not changes the overall complexity of the $ParallelCandidateReductionOperator$ function. Computing the length of X and Y (step 11) requires $O(1)$, and determining the candidate elements and their weights requires in the worst case $O(k)$.

Indeed, in Algorithm 2.2 we slightly modify the original *Frequent* algorithm. We take advantage of the fact that, for each element appearing in X, we can process in at most two steps its occurrences, which are stored in Y. We also note that a given element may appear in X at most twice, since it can appear at most once in both $local_i$ and $local_j$, owing to the fact that $local_i$ and $local_j$ are the output of the $Frequent$ algorithm.

The algorithm works exactly as the original one, but in our case, when we process an element in X, we can increment a counter by the corresponding amount of occurrences stored in Y. All the steps inside the *while* loop can be done in $O(1)$ time. In particular, decrementing all the counters and keeping track of the minimum among the counters can be done in $O(1)$ by using the same data structure described in [8] (counters are stored in sorted order using a differential encoding).

We now prove the following statement.

Lemma 4.1. *The while loop of step 13 is executed at most twice.*

Proof. Either the weight of the current element is reset to zero in the first iteration, when there is an available counter in charge of monitoring it, or, if no counter is available, all the counters and the weight are decremented by m. Therefore, if m is equal to the weight of the current element, then the weight of the current element is reset to zero in the first iteration. Otherwise, at least one counter has been reset to zero after the decrement operation and will be available to monitor the element in the second iteration. □

Consequently, since each iteration requires in the worst case at most $O(1)$ time, and by Lemma 4.1 the *while* loop is executed at most $O(1)$ times, it follows that the overall complexity of the *while* loop is at most $O(1)$ time. The *for* loop of step 12 is executed, as already stated, at most $2(k-1) = O(k)$ times. Therefore, the complexity of Algorithm 2.2 is at most $O(k)$ to determine the candidate elements and their corresponding weights, which are then passed on to another processor to continue the reduction in parallel.

Owing to the fact that the *ParallelCandidateReductionOperator* function is called in each step of the parallel reduction and its complexity is $O(k)$, the overall complexity of the parallel reduction executed in step 4 of Algorithm 2.1 is $O(k \log p)$ (using, for instance, a Binomial Tree [25] or even a simpler Binary Tree). Therefore, the parallel complexity of Algorithm 2.1 to determine k–majority candidates is $O(n/p + k \log p)$. Since the class of applications we are targeting is characterized by the property $k = O(1)$, we are now in the position to state the following Lemma.

Lemma 4.2. *Algorithm 2.1 is cost–optimal for $k = O(1)$.*

Proof. Cost–optimality requires by definition $p\ T_p = T_1$ asymptotically, 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 $O(n)$, and the parallel complexity of our algorithm when $k = O(1)$ is $O(n/p + \log p)$. It follows from the definition that the algorithm is cost–optimal for $n = \Omega(p \log p)$. □

We now analyze isoefficiency and scalability. The sequential algorithm has complexity $O(n)$; the parallel overhead is $T_o = p\ T_p - T_1$. In our case, $T_o = p\ (n/p + k \log p) - n = k\ p \log p$. The isoefficiency relation [26] is then $n \geq k\ p \log p$). Finally, we derive the scalability function of this parallel system [27].

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; $C\ p$ represents 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, $M(f(p))/p = O(k \log p)$ with $f(p)$ given by the isoefficiency relation. When the property $k = O(1)$ holds, correspondingly the scalability function becomes $O(\log p)$.

We analyze now Algorithm 2.3 and derive its parallel complexity. The initial domain decomposition done in steps 1–2 requires $O(1)$ time. Then, we verify the number of occurrences of the global candidate found in each sub–array. In step 3 we first construct an hash table containing the candidate elements by using Cuckoo Hashing [28], and then we do a linear scan, checking if the current element being examined belongs to the Cuckoo hash table in $O(1)$. Indeed, this data structure provides us with guaranteed *worst case constant lookup time*; building the hash table requires expected $O(k)$ time. Therefore, the complexity of this step is, $O(k + n/p)$. All the processors engage in a parallel sum reduction (step 4), whose complexity is $O(k \log p)$, to determine the total number of occurrences for all the candidate elements present in the whole input array.

The processor whose rank is zero determines sequentially the set of k–majority elements. We initialize the $frequent$ set in $O(1)$ time as an empty set in step 6. The for loop of steps 7–10 simply inserts an element into $frequent$ if its occurrences are greater than n/k. Finally, we output the $frequent$ set in step 11.

It follows that the overall *average parallel complexity* of the verify algorithm is $O(n/p + k \log p)$. However, for $k = O(1)$, this reduces to *worst–case* $O(n/p + \log p)$, since (i) we do not need to use a Cuckoo hash table to determine the occurrences of the candidate elements in a local sub–array, (ii) the parallel reduction requires $O(\log p)$ time and (iii) constructing the $frequent$ set can be done in $O(1)$ time. Consequently, owing to the fact that for $k = O(1)$ Algorithm 2.3 shares the same complexity of Algorithm 2.1, its cost–optimality follows immediately.

Acknowledgment

The research of M. Cafaro and P. Tempesta has been supported respectively by CMCC, Italy and by the grant FIS2008–00260, Ministerio de Ciencia e Innovación, Spain.

REFERENCES

1. Syropoulos A. Mathematics of multisets. *In Multiset Processing: Mathematical, computer science, and molecular computing points of view, LNCS 2235*, Springer–Verlag, 2001; 347–358.
2. Boyer R, Moore J. Mjrty – a fast majority vote algorithm. *Technical Report 32*, Institute for Computing Science, University of Texas, Austin 1981.
3. Boyer R, Moore JS. Mjrty – a fast majority vote algorithm. *Automated Reasoning: Essays in Honor of Woody Bledsoe, Automated Reasoning Series, Kluwer Academic Publishers, Dordrecht, The Netherlands.* 1991; 105–117.
4. Fischer M, Salzberg S. Finding a majority among n votes: Solution to problem 81–5. *J. of Algorithms* 1982; (3):376–379.
5. Fang M, Shivakumar N, Garcia-Molina H, Motwani R, Ullman JD. Computing iceberg queries efficiently. *Proceedings of the 24th International Conference on Very Large Data Bases, VLDB. Morgan–Kaufmann, San Mateo, Calif.*, 1998; 299–310.
6. Beyer K, Ramakrishnan R. Bottom–up computation of sparse and iceberg cubes. *Proceedings of the ACM SIGMOD International Conference on Management of Data. ACM, New York*, 1999; 359–370.
7. Gibbons PB, Matias Y. Synopsis data structures for massive data sets. *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science: Special Issue on External Memory Algorithms and Visualization, vol. A.*, American Mathematical Society: Boston, MA, USA, 1999; 39–70.

8. Demaine ED, López-Ortiz A, Munro JI. Frequency estimation of internet packet streams with limited space. *ESA*, 2002; 348–360.
9. Estan C, Varghese G. New directions in traffic measurement and accounting. *IMW '01: Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, ACM: New York, NY, USA, 2001; 75–80, doi:http://doi.acm.org/10.1145/505202.505212.
10. Pan R, Breslau L, Prabhakar B, Shenker S. Approximate fairness through differential dropping. *SIGCOMM Comput. Commun. Rev.* 2003; **33**(2):23–39, doi:http://doi.acm.org/10.1145/956981.956985.
11. Brin S, Motwani R, Ullman JD, Tsur S. Dynamic itemset counting and implication rules for market basket data. *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, ACM: New York, NY, USA, 1997; 255–264, doi:http://doi.acm.org/10.1145/253260.253325.
12. Charikar M, Chen K, Farach-Colton M. Finding frequent items in data streams. *ICALP '02: Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, Springer–Verlag: London, UK, 2002; 693–703.
13. *Computational Linguistics and Intelligent Text Processing, 7th International Conference, CICLing 2006, Mexico City, Mexico, February 19-25, 2006, LNCS 3878*. Springer–Verlag, 2006.
14. Zipf GK. *Selected Studies of the Principle of Relative Frequency in Language*. Cambridge University Press, 1932.
15. Mandelbrot B. Information theory and psycholinguistics: A theory of word frequencies. *Language: selected readings, edited by R.C. Oldfield and J.C. Marchall, Penguin Books*. 1968.
16. Mouillot D, Lepretre A. Introduction of relative abundance distribution (rad) indices, estimated from the rank-frequency diagrams (rfd), to assess changes in community diversity. *Environmental Monitoring and Assessment* 2000; **63**(2):279–295.
17. Misra J, Gries D. Finding repeated elements. *Sci. Comput. Program.* 1982; **2**(2):143–152.
18. Adel'son–Vel'skii GM, Landis EM. An algorithm for the organization of information. *Soviet Mathematics Doklady* 1962; **3**:1259–1262.
19. Karp RM, Shenker S, Papadimitriou CH. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.* 2003; **28**(1):51–55, doi:http://doi.acm.org/10.1145/762471.762473.
20. Cormode G, Hadjieleftheriou M. Finding the frequent items in streams of data. *Commun. ACM* 2009; **52**(10):97–105, doi:http://doi.acm.org/10.1145/1562764.1562789.
21. Manku GS, Motwani R. Approximate frequency counts over data streams. *In VLDB*, 2002; 346–357.
22. Metwally A, Agrawal D, Abbadi AE. Efficient computation of frequent and top-k elements in data streams. *International Conference on Database Theory*, 2005; 398–412.
23. Cormode G, Muthukrishnan S. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms* 2005; **55**(1):58–75, doi:http://dx.doi.org/10.1016/j.jalgor.2003.12.001.
24. Mehlhorn K. *Data Structures and Algorithms 1: Sorting and Searching*, *Monographs in Theoretical Computer Science. An EATCS Series*, vol. 1. Springer, 1984.
25. Cormen TH, Leiserson CE, Rivest RL, Stein C. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.
26. Grama A, Gupta A, Kumar V. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Parallel and Distributed Technology* Aug 1993; **1**(3):12–21.
27. Quinn MJ. *Parallel Programming in C with MPI and OpenMP*. McGraw–Hill, 2003.
28. Pagh R, Rodler FF. Cuckoo hashing. *J. Algorithms* 2004; **51**(2):122–144, doi: http://dx.doi.org/10.1016/j.jalgor.2003.12.002.