Finding the Majority Element in Parallel

Massimo Cafaro[a,b,*], Piergiulio Tempesta[c,**]

[a] *University of Salento, Lecce, Italy*
Dept. of Engineering for Innovation - via Per Monteroni, 73100 Lecce, Italy
[b] *Euro Mediteranean Center for Climate Change*
73100 Lecce, Italy
[c] *Departamento de Física Teórica II, Facultad de Físicas,*
Universidad Complutense, 28040 – Madrid, Spain

Abstract

Given an array A of n elements, the *majority element* is an element occurring in A more than $n/2$ times. The majority problem requires finding the majority element whenever it exists. In this paper we solve the majority problem, by proposing a cost–optimal parallel algorithm, in the context of message–passing architectures. We present an in–depth theoretical analysis of the algorithm and formally prove its correctness.

Keywords:
Majority Element, message-passing

1. Introduction

In this work, we propose a deterministic, cost–optimal parallel algorithm for the majority problem. To the best of our knowledge, this is the first correct message–passing parallel algorithm solving it completely, i.e. for any finite input array. The mathematical setting we adopt is that of multiset theory [1], since it provides a natural and elegant language for handling the technical aspects of the proposed solution. In order to state our problem in full generality, we recall some basic definitions.

*Principal Corresponding author
**Corresponding author
Email addresses: `massimo.cafaro@unisalento.it` (Massimo Cafaro), `p.tempesta@fis.ucm.es` (Piergiulio Tempesta)

Definition 1. *A multiset is a pair (A, m), where A is some set, called the underlying set of elements, and $m : A \rightarrow \mathbb{N}$ is a function.*

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. $m(x, A) \equiv m(x) = Card(\{i : A[i] = x\})$.

Definition 2. *Given a* multiset *A of n elements, the majority element μ is the unique element whose* multiplicity *$m(\mu)$ is such that $m(\mu) \geq \left\lfloor \frac{n}{2} \right\rfloor + 1$.*

Our problem can then be formulated as follows.

Definition 3. *Majority problem.*
Input: An array A of n numbers.
Output: The singleton $\{\mu\}$ or $\emptyset$.

Finding the majority element, besides being interesting from a theoretical perspective, is also of practical use, for instance in all of the cases, such as electronic voting, where a quorum of more than half of all of the preferences received is required for a candidate to win. Other examples include analyzing the statistical properties of a TCP/IP packet stream arriving at an Internet router to determine the heaviest user, or the most popular web site, or, in the case of QoS (Quality of Service) Internet traffic, to determine whether or not a particular DiffServ Service Class [2] prevails etc. It appears natural to consider a parallel processing of the input instances, since they come from different and independent sources.

A sequential version of the majority problem has been studied in [3], [4], [5]. In this paper, we design a parallel algorithm, that provides a fast and elegant solution of such problem. This article is organized as follows. Our algorithm is presented in Section 2. We prove its correctness in Section 3, and its cost–optimality in Section 4. Section 5 is devoted to the comparison of our results to some related work. In particular, we discuss the Lei and Liaw algorithm [6]. We conclude the paper by showing that it is not always correct, by providing a counterexample.

2. Our Algorithm

The pseudocode of Algorithm 2.1 describes our parallel majority algorithm. We assume that the array 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 PARALLEL–MAJORITY(A, n, p), where A is the input array, n the length of A and p the number of processors we use in parallel.

Algorithm 2.1: Parallel Majority algorithm

```
   Input: $A$, an array; $n$, the length of $A$; $p$, the number of processors
   Output: a set containing the majority element of $A$ if it exists,
           otherwise an empty set
   /* The $n$ elements of the input array $A$ 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  $left \leftarrow \lfloor id\ n/p \rfloor$;
2  $right \leftarrow \lfloor (id + 1)\ n/p \rfloor - 1$;
   /* determine a local candidate and its weight */
3  $c, w \leftarrow$ Boyer-Moore(A, left, right);
   /* determine the global candidate for the whole array */
4  $gc \leftarrow$ ParallelCandidateAllReduction(c, w);
   /* determine the number of occurrences of the global
      candidate */
5  $m \leftarrow$ Occurrences(A, gc, left, right);
   /* determine the number of occurrences for the whole array
      */
6  $count \leftarrow$ ParallelSumReduction(m);
   /* we assume here that the processor with rank 0 holds the
      final result of the parallel reduction */
7  if $id == 0$ then
8      if $count \geq \lfloor \frac{n}{2} \rfloor + 1$ then
9          return $\{gc\}$;
10     else
11         return $\emptyset$;
```

The parallel majority algorithm works as follows. The initial domain decomposition is done in steps 1–2. Each processor determines the indices

Algorithm 2.2: Boyer–Moore algorithm

```
Input: $A$, an array; $left$, the first element of $A$ to be processed; $right$,
       the last element of $A$ to be processed
Output: $c$, a majority candidate and $w$, its corresponding weight
counter ← 0;
for $i \leftarrow left$ to $right$ do
    if $counter == 0$ then
        $candidate \leftarrow A[i]$;
        $counter \leftarrow 1$;
    else
        if $candidate == A[i]$ then
            $counter \leftarrow counter + 1$;
        else
            $counter \leftarrow counter - 1$;
end
return $candidate$, $counter$;
```

of the first and last element related to its block, by applying a simple block distribution, in which the k–th processor is responsible for the sub–array A_k, $k = 0, \ldots, p-1$, consisting of either $\lfloor n/p \rfloor$ or $\lceil n/p \rceil$ elements. Since each process receives as input the whole array A, there is no need to use message–passing to perform the initial domain decomposition. Then, each processor applies the (initial phase of the) well–known Boyer–Moore algorithm [3], [4] shown in the text as Algorithm 2.2.

We introduce some useful definitions.

Definition 4. *A local majority candidate $c_k \in A_k$ for $k = 0, \ldots, p-1$ is the output of the Boyer–Moore algorithm, given by the processor k operating on the input A_k.*

Definition 5. *The (global) majority candidate $c \in A$ is the output of the ParallelCandidateReductionOperator.*

In step 3, each processor determines its local majority candidate (i. e. an element that could be the majority element in the sub–array A_k) and its related weight. The algorithm then determines a global majority candidate (i.e an element $c \in A$ that may be the majority element), and proceeds to

```
Function ParallelCandidateReductionOperator($c_1, m_1, c_2, m_2$)
   Input: $c_1$, a candidate element; $m_1$, the occurrences of $c_1$; $c_2$, a
          candidate element; $m_2$, the occurrences of $c_2$
   Output: $c$, a candidate element and $m$, its occurrences
   /* Check if the two candidates are equal */
   if $c_1 == c_2$ then
       $c \leftarrow c_1$;
       $m \leftarrow m_1 + m_2$;
   else
       /* Check the candidates' occurrences */
       $d \leftarrow m_1 - m_2$;
       switch $d$ do
           case $d < 0$
               $c \leftarrow c_2$;
               $m \leftarrow m_2 - m_1$;
           case $d > 0$
               $c \leftarrow c_1$;
               $m \leftarrow d$;
           case $d == 0$
               $c \leftarrow DUMMY - CANDIDATE$;
               $m \leftarrow 0$;
       end
   return $c, m$;
```

verify whether the candidate actually is the majority element or not. In step 1 of Algorithm 2.2 the integer variable *counter* is initialized to zero. Then, a linear scan of the array (step 2) processes in turn each input element, updating as needed the majority candidate and the counter. If the counter is zero (step 3), then the algorithms sets as the majority candidate the element being examined and increments the counter (steps 4–5). Otherwise (step 6), the counter is greater than zero. In this case if the majority candidate is equal to the element being examined (step 7), the counter is incremented (step 8), otherwise it is decremented (steps 9–10). Finally, the algorithms returns in step 12 the majority candidate determined and its weight, i.e. the value of the counter.

Once all the majority candidates $c_k, k = 0, \dots, p-1$ have been found, in Algorithm 2.1 the pair (c_k, w_k) is used as input for the parallel all reduction of step 4. Its purpose is to determine the global majority candidate c for the whole array. The ALL–REDUCTION communication is functionally equivalent to a reduction followed by a broadcast, and allows each processor to receive the result of the reduction. This step is carried out using the $ParallelCandidateReductionOperator$ function during the reduction. Then, each processor determines the number of occurrences of c in its sub–array in step 5. Finally, the processors engage in a parallel sum reduction in step 6 to determine *count*, the total number of occurrences of c. Assume that at the end of the parallel sum reduction the processor whose rank is zero holds the result of the parallel reduction step. Then, steps 8–11 carried out by the rank–zero processor either output the singleton set, containing the majority element $\mu = c$ or the empty set, depending on the test $count \geq \left\lfloor \frac{n}{2} \right\rfloor + 1$.

3. Proof of Algorithm's Correctness

In this Section we prove that our algorithm is correct, i.e. that it provides the correct majority element, when it exists, by combining globally the local results coming from all sub–arrays. Since a false majority element is discarded by the verification procedure, it is enough for the algorithm to be correct that it recognizes μ as the majority element, in all of the input instances where it exists.

Our strategy is based on the following Lemma [7].

Lemma 1. *Given an array X of n elements, repeatedly deleting pairs of distinct elements from it until no longer possible leads to a 2–reduced multiset,* *i.e. a multiset containing only elements equal to the majority candidate or the empty set.*

PROOF. The maximum number of times we can delete a pair of distinct elements from the array is bounded by $\lfloor n/2 \rfloor$, owing to the fact that, once we reach this limit, the array contains less than 2 elements. It follows that, when the array no longer contains distinct elements, is either empty (in which case no majority element exists) or contains the majority candidate c with multiplicity $m(c) \geq 1$. $\square$

Let c_k be the local majority candidate in the sub–array A_k, consisting of e_k elements, analyzed by the processor k, with $k = 0, \ldots, p-1$. Each process is labeled by an integer variable called process *rank*, denoted by id and satisfying the inequalities $0 \leq id \leq p-1$. Let $m_k \equiv m(c_k)$. From the block distribution used for the initial domain decomposition, it follows that

$$e_k = \lfloor (id + 1)\ n/p \rfloor - \lfloor id\ n/p \rfloor . \qquad (1)$$

When p divides n, eq. (1) reduces to $e_k = n/p$.

The Boyer–Moore algorithm uses a counter to determine the majority candidate. By using the counter value as the weight of the local candidate c_k just determined, each processor correctly constructs a 2–reduced multiset.

We prove the previous statements in the subsequent Lemmas and Theorems. We begin by establishing that the algorithm correctly determines the local 2–reduced multiset of each processor. Then we proceed to prove that the parallel candidate reduction correctly determines the majority candidate for the global 2–reduced multiset. The correctness of algorithm 2.2 has already been proved by Boyer and Moore in [3], [4]; here we show that, by construction, their algorithm is equivalent to determining a local 2–reduced multiset.

Lemma 2. *The $Boyer - Moore$ algorithm is equivalent to constructing the local 2–reduced multiset of a processor in the order in which the input elements are read.*

PROOF. The algorithm assigns a counter a_k to each subarray A_k. The value of the counter is incremented by one if the element being considered is equal to the current majority candidate associated to the counter, otherwise it is decremented by one. This last operation is equivalent to deleting a couple of distinct elements, whenever is possible. Consequently, the final value $a_{k,f}$ of the counter is equal to the occurrences of the remaining element, after the deletion process ends. The integer numbers $a_{k,f}$, for $k = 0, \dots p-1$, depend a priori on the ordering of the stream of data. However we will see that this aspect, due to Lemma 1, does not affect the correctness of the global procedure. □

Referring to the pseudocode for the *ParallelCandidateReductionOperator* function, steps 1–3 deal with the case $c_1 = c_2$ returning c_1 as the global candidate with occurrences given by the sum of o_1 and o_2. The remaining steps cover the cases $c_1 \neq c_2$. When $m_2 > m_1$ (steps 7–9) we return c_2 as a global candidate with occurrences given by the difference $m_2 - m_1$. The case $m_1 > m_2$ (steps 10–12) is symmetric: we return c_1 as a global candidate with occurrences given by the difference $m_1 - m_2$. The last case occurs when $m_1 = m_2$ (steps 13–15). In this case, an arbitrary element could be returned as a global candidate, along with zero occurrences.

We are now in a position to state the following theorem:

Theorem 1. *The parallel majority algorithm is correct.*

Proof. If the initial stream of data does not contain the majority element, the determined majority candidate will be correctly discarded in the final verification phase of the algorithm. Therefore, in what follows we assume that the initial stream of data contains the majority element. The Algorithm is correct if it is able to detect it. In order to determine the global majority candidate, according to Lemma 2, the parallel majority algorithm operates by deleting pairs of distinct elements, whenever is possible. Indeed, in the parallel reduction step, we add the occurrences of equal elements since they can not be paired (they are not distinct elements), and subtract the occurrences of elements that differ, and therefore can be paired.

In all sub–arrays A_k where a local majority element μ_k is not present, the final value $a_{k,f}$ of the counter a_k is either zero, if n/p is even, or is equal to the multiplicity of the element remaining after the deletion process, which is the local candidate c_k determined. Observe that the function used to reduce candidates in parallel combines intermediate results, by further removing distinct pairs across sub–arrays. The values of the counters depend on the ordering of the sub–arrays, and more generally on the way we distribute data between our processors. However, the correctness of the algorithm is guaranteed, by taking into account the following facts.

i) Every possible partition of the data stream into different processors is allowed. Indeed, the operation of distributing elements in different sub–arrays is equivalent to choosing a specific ordering of the data stream.

ii) Lemma 1 is valid *independently of the order* we use to remove distinct elements.

Consequently, in the last step of the parallel reduction, if the majority element exists, it must coincide with the global majority candidate being determined, and its associated weight, computed by the parallel reduction, never goes to zero. The ordering would affect the actual value of the weight being computed, but not the fact that it is a positive integer. This is sufficient for a majority element to be detected correctly. $\square$

This Theorem is the main result of this paper, jointly with the Theorem in the next Section, in which we prove that our algorithm is cost–optimal.

4. Analysis of the Algorithm

In this Section, we derive the parallel complexity of the algorithm. At the beginning of the PARALLEL–MAJORITY 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 $O(1)$. Determining a local majority candidate and its weight in step 3 using the Boyer–Moore algorithm reduces to scanning the array and incrementing or decrementing a counter variable, so that this step requires $O(n/p)$ time.

The *ParallelCandidateReductionOperator* function is used internally by the all reduction step. Since this function is called in each step of the parallel all reduction and its complexity is $O(1)$, the overall complexity of the parallel all reduction step is $O(\log\ p)$ (using, for instance, a Binomial Tree [8] or even a simpler Binary Tree). Therefore, the parallel complexity of the initial phase in which the algorithm determines a global candidate is $O(n/p + \log\ p)$.

The remaining steps verify the number of occurrences of the global candidate found; this reduces to a linear scan in time $O(n/p)$ to determine the local number of occurrences in each processor and a parallel sum reduction whose parallel complexity is $O(\log\ p)$. The final check done in steps 7–11 by the processor whose rank is zero requires $O(1)$ time. It follows that the parallel complexity of the final phase is $O(n/p + \log\ p)$. Since the parallel complexity of both the initial and final phases is the same, the entire algorithm's complexity is $O(n/p + \log\ p)$.

We are now in the position to state the following Theorem:

Theorem 2. *The algorithm is cost–optimal.*

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

We proceed with the analysis of isoefficiency and scalability. The sequential algorithm has complexity $O(n)$; the parallel overhead is $T_o = p\ T_p - T_1$. In our case, $T_o = p\ (n/p + \log\ p) - n = p\ \log\ p$. The isoefficiency relation [9] is then $n \geq p\ \log\ p)$. Finally, we derive the scalability function of this parallel system [10].

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 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.

Finally, it is worth noting here that another possible parallel algorithm could take into account in each processor the occurrences of other possible majority candidates besides the local one. One way to achieve this is exchanging the p local majority candidates through an ALL–GATHER communication, determining the local and global occurrences of each one. However, since there can be up to p distinct majority candidates, the parallel complexity of this algorithm would be higher than the corresponding sequential counterpart. Indeed, this algorithm would require $O(p\ n/p + p) = O(n + p)$. Our algorithm instead avoids, as shown, additional communications.

5. Related Work

The only parallel algorithm in the context of message–passing architectures for the majority problem we are aware of, has been proposed by Lei and Liaw [6]. All the other references we are aware of, either refer to a different parallel model (e.g., shared memory with reference to the PRAM model) or deal with a different statement of the problem, although similar to ours. Among these, we recall the parallel algorithms for m–out–of–n threshold voting [11]; distributed voting algorithms in the context of distributed database systems include [12], and weighted majority algorithms resilient to noisy data have been proposed in the context of machine learning [13].

The algorithm by Lei and Liaw [6] is similar to ours, with the following differences.

1. Each processor determines its local majority candidate by utilizing a linear time algorithm given by Lei and Liaw; in this paper we use instead the initial phase of the well–known Boyer–Moore algorithm. Using the Boyer–Moore algorithm to determine local majority candidates provides a slightly faster parallel algorithm; the Lei and Liaw algorithm also requires additional $O(n)$ space, owing to the fact that the input array needs to be copied into a temporary work array.
2. Their algorithm uses the number of occurrences associated to the local majority candidate determined, instead of the number of elements belonging to the local 2–reduced multiset as input for the parallel reduction. This is enough to claim that their algorithm is not correct: we now show this fact by providing a carefully crafted input as a counterexample.

Lemma 3. *Given an array A of n elements, the algorithm of Lei and Liaw fails to provide the correct output for all of its inputs.*

Proof. It is enough to show an input for which, while the majority element μ exists, the algorithm outputs the empty set. Suppose the array A contains 13 elements and there are 4 processors:

$A = [1, 2, 1, 2, 1, 1, 2, 2, 2, 2, 2, 2, 1, 1]$. Therefore, the array A contains a majority element, namely 2, since this element occurs 7 times in A.

The initial domain decomposition distributes the elements to the processors as follows:

$p_0 : [1, 2, 1]$; $p_1 : [2, 1, 1]$; $p_2 : [2, 2, 2]$; $p_3 : [2, 2, 1, 1]$.

The processors determine the following pairs ($localCandidate, Occurrences$): $p_0 : (1, 2)$; $p_1 : (1, 2)$; $p_2 : (2, 3)$; $p_3 : (dummy, 0)$.

In order to determine a global majority candidate, the parallel reduction works as follows.

Step 1: p_0 *vs* p_1: since the majority candidate 1 is the same in both pairs, we sum the occurrences, and the result is the pair (1, 4); p_2 *vs* p_3: here the majority candidates are such that $2 \neq dummy$ and $3 > 0$, so that the result is the pair (2, 3);
Step 2: the pairs (1, 4) and (2, 3) are compared; since $1 \neq 2$ and $4 > 3$, the result is the pair (1, 1).

Therefore, their algorithm determines 1 as global majority candidate. When this majority candidate is verified, the algorithm determines that its occurrences are less than or equal to $n/2$ and, incorrectly, outputs the empty set. $\square$

The reason why the algorithm of Lei and Liaw is not always correct is that their algorithm does not take into account the possibility that the majority element may occur in different partitions. Therefore, when applying the parallel reduction to determine the global majority candidate, it is actually possible, as shown, to produce a wrong output. Indeed, the reduction operation, used to find in parallel a global majority candidate is correct, but its input is not. By using as input pairs ($localCandidate, Occurrences$), it fails to correctly consider that the occurrences of the element 2 in the block assigned to processor p_3, when summed to the ones in the blocks of p_0, p_1 and p_2 can achieve the majority. This, in turn, leads to a wrong output.

Let us now show how our algorithm works on the same input. The initial domain decomposition distributes the elements to the processors identically: $p_0 : [1, 2, 1]$; $p_1 : [2, 1, 1]$; $p_2 : [2, 2, 2]$; $p_3 : [2, 2, 1, 1]$. The processors determine the following pairs ($localCandidate, weight$): $p_0 : (1, 1)$; $p_1 : (1, 1)$; $p_2 : (2, 3)$; $p_3 : (2, 0)$. In order to determine the global majority candidate, the parallel reduction works as follows.

Step 1: p_0 *vs* p_1: since the majority candidate 1 is the same in both pairs, we sum the weights, and the result is the pair (1, 2); p_2 *vs* p_3: here the majority candidate 2 is the same in both pairs too, and the result is the pair (2, 3).
Step 2: the pairs (1, 2) and (2, 3) are compared; since $1 \neq 2$ and $3 > 2$, the result is the pair (2, 1).

Therefore, our algorithm determines 2 as the global majority candidate

c. When this majority candidate is verified, the algorithm determines that its occurrences $m(c) \geq \lfloor \frac{n}{2} \rfloor + 1$ and, correctly, outputs the singleton set $\{2\}$.

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] A. Syropoulos, Mathematics of multisets, in: In Multiset Processing: Mathematical, computer science, and molecular computing points of view, LNCS 2235, Springer-Verlag, 2001, pp. 347–358.

[2] J. Babiarz, K. Chan, F. Baker, Configuration guidelines for diffserv service classes, RFC 4594, Internet Engineering Task Force (2006).

[3] R. Boyer, J. Moore, Mjrty - a fast majority vote algorithm, Tech. Rep. 32, Institute for Computing Science, University of Texas, Austin (1981).

[4] R. Boyer, 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, 1991, pp. 105–117.

[5] M. Fischer, S. Salzberg, Finding a majority among n votes: Solution to problem 81-5, J. of Algorithms (3) (1982) 376–379.

[6] C.-L. Lei, H.-T. Liaw, Efficient parallel algorithms for finding the majority element, J. Inf. Sci. Eng. 9 (2) (1993) 319–334.

[7] J. Misra, D. Gries, Finding repeated elements, Sci. Comput. Program. 2 (2) (1982) 143–152.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, Second Edition, The MIT Press, 2001.

[9] A. Grama, A. Gupta, V. Kumar, Isoefficiency: Measuring the scalability of parallel algorithms and architectures, IEEE Parallel and Distributed Technology 1 (3) (1993) 12–21.

[10] M. J. Quinn, Parallel Programming in C with MPI and OpenMP, McGraw-Hill, 2003.

[11] B. Parhami, Parallel threshold voting, Comput. J. 39 (8) (1996) 692–700.

[12] N. R. Adam, A new dynamic voting algorithm for distributed database systems, IEEE Trans. Knowl. Data Eng. 6 (3) (1994) 470–478.

[13] N. Littlestone, M. K. Warmuth, The weighted majority algorithm, in: SFCS '89: Proceedings of the 30th Annual Symposium on Foundations of Computer Science, IEEE Computer Society, 1989, pp. 256–261.