Parallel Algorithms for m-out-of-n Threshold Voting

Behrooz Parhami

Department of Electrical and Computer Engineering
University of California
Santa Barbara, CA 93106-9560, USA
E-mail: parhami@ece.ucsb.edu

Abstract: Voting on large collections of input objects is becoming increasingly important in data fusion, signal and image processing, and distributed computing. To achieve high speed in voting, the multiple processing resources typically available in such applications should be utilized; hence the need for parallel voting algorithms. We develop efficient parallel algorithms for threshold voting which generalize and extend previous work on both sequential threshold voting and parallel majority voting. We show how a well-known $O(n)$-time sequential algorithm for m-out-of-n voting can be parallelized through a simple divide-and-conquer strategy. When $m = \theta(n)$, the resulting algorithm has $O(\log^2 n)$ time complexity on PRAM and hypercube computers and optimal $O(n^{1/k})$ complexity on a k-dimensional mesh-connected architecture. We also analyze the time complexity of the algorithm in the case of $m = o(n)$ and for certain weighted threshold voting schemes.

Keywords: Data fusion, Dependable computation, Distributed processing, Majority voting, Multi-channel computation, Parallel complexity, Voting schemes.

1. Introduction

Voting has long been an important operation in the fusion of data originating from multiple sources and in the realization of ultrareliable digital systems based on multi-channel computation [20]. In data fusion, voting is a possible way of combining diverse data provided by multiple sources (such as sensors) whose outputs may be erroneous, incomplete, tardy, or totally missing. In ultrareliable systems, voting is required whether the multiple computation channels consist of redundant hardware units, diverse program modules executed with common data on the same basic hardware, identical hardware and software with diverse data, or hybrid combinations of hardware/ program/data redundancy and/or diversity.

Whereas traditional applications of voting have been limited to processing a small number of simple input objects (typically bits or numerical words), newer applications involve both larger input sets and more complex input objects. Examples include fusion of data originating from a large number of sensors [8], [20], [21], image processing filters which smooth digital pictures by voting on neighborhoods [2], and certain configuration control, bookkeeping, and diagnostic functions in parallel and distributed systems [5], [12], [22].

Hence, the efficiency of voting algorithms (both time and space complexity) are becoming important. Previous works on this aspect of voting have been limited to efficient sequential majority voting algorithms [1], [7], [13], sequential threshold voting [3], and parallel majority voting [10]. In all cases, simple unweighted voting has been assumed. We have previously extended these efficient threshold voting algorithms to the weighted case in the context of a comprehensive study of voting schemes [15], [16], [17], [18]. In this paper, we generalize and extend the above to efficient parallel m-out-of-n threshold voting algorithms.

The rest of this paper is organized as follows. In Section 2, we define the scope of the voting schemes that are of interest in this study. Section 3 contains a review of a voting algorithm that can be used for weighted or unweighted threshold voting with equal efficiency. In Section 4, we show how the unweighted m-out-of-n version of the above algorithm can be parallelized based on a simple divide-and-conquer strategy. Sections 5 and 6 contain discussions of the complexity of our parallel m-out-of-n threshold voting algorithm for $m = \theta(n)$ and $m = o(n)$, respectively. Section 7 deals with modifications required in the algorithm, and the associated complexity issues, for weighted voting. We conclude, in Section 8, with a summary of our contributions and directions for further research.

2. Threshold Voting Schemes

In order to facilitate and systematize the study of voting schemes, we have previously categorized them according to implementation in hardware or software (voting *networks* [14] or voting *routines*) and based on the size and structure of the input object space (see Figure 1). A voting *algorithm* [18] specifies how the voting result is obtained from the input data and may be the basis for designing a voting network or a voting routine.

As shown in Figure 1, the input objects to be voted upon can be atomic or composite. Composite objects, consisting implicitly or explicitly structured collections of atomic objects, have not received due attention in previous works on voting. With atomic objects, the input object space can be small or large.

0-7803-3529-5/96/$5.00 © 1996 IEEE

For small object spaces, further classification is unimportant, as they always lead to very simple and efficient voting algorithms. For large object spaces, whether or not a distance metric can be defined, and as a special case, if the objects can be ordered, is important. Finally, for an unordered space, voting algorithms tend to be less complex if the notion of "support", as discussed in Definition 2.1 below, is transitive (i.e., if X supports Y and Y supports Z together imply that X supports Z).

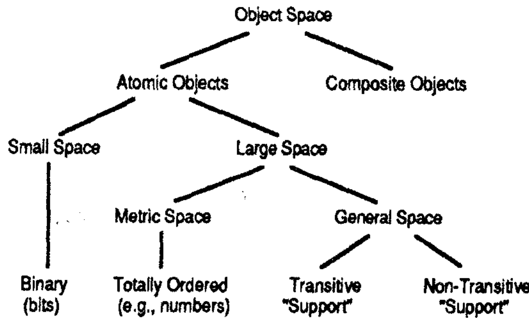

Figure 1. Voting schemes classified according to the object space size and structure.

Figure 2 shows an example of composite data objects that might be used in voting. The objects x_1, x_2, x_3, x_4 depicted in Figure 2 are infinite sets of numbers defined by the four closed intervals $[l_1, h_1]$, $[l_2, h_2]$, $[l_3, h_3]$, $[l_4, h_4]$ on the real line. The voting algorithm may depend on the semantics attached to these intervals and on application requirements. For example, if the intervals are considered as different views of the safe operating range for some physical parameter in a critical system, then the interval $[l_2, h_4]$ may be taken as the voting outcome in view of its unanimous designation as being safe. If one of the evaluators (combination of sensors and decision logic) fails so that its corresponding interval is "way off", then majority consensus can still be reached.

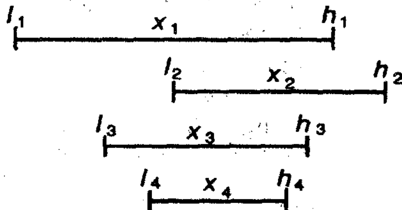

Figure 2. Voting with composite data objects represented as intervals on the real line.

To accommodate all voting schemes of interest, including those dealing with composite data objects, we present the following general definition of weighted threshold voting.

2.1. Definition (weighted threshold voting): Given n input data objects $x_1, x_2, \ldots, x_n$, and their associated non-negative real votes (weights) $v_1, v_2, \ldots, v_n$, with $\sum_{i=1}^{n} v_i = V$, it is required to compute the output y and its vote w such that y is "supported by" a number of input data objects with votes totaling w, where w satisfies a condition associated with the desired voting subscheme; e.g., $w > V/2$ for majority voting, $w > 2V/3$ for Byzantine voting, and $w \geq t$ for t-out-of-V (generalized m-out-of-n) voting. Note that for t-out-of-V voting with $t \leq V/2$, or its special case of m-out-of-n voting with $m \leq n/2$, the output may be non-unique. In such cases, arbitrary selection of y from among valid outputs is assumed. ∎

The term "supported by", which accounts for the generality of Definition 2.1, can be defined in several ways, leading to different voting schemes. With exact voting, an input x_i supports y iff $x_i = y$. With inexact voting, approximate inequality ($\cong$) is defined in some suitable way (e.g. by providing a comparison threshold ε in the case of numerical values or, more generally, a distance measure d in a metric space) and x_i is said to support y iff $x_i \cong y$. With approval voting, y must be a subset of the approved set of values that x_i defines using some suitable encoding of the sets (e.g., bit-vectors, lists, or intervals).

We get m-out-of-n voting as a special case of the weighted threshold voting of Definition 2.1 by letting $v_1 = v_2 = \ldots = v_n = 1$ and $t = m$. Majority voting is obtained if we further restrict m to $(n + 1)/2$. For $m > n/2 + 1$, stronger agreement is required (super-majority), whereas in the case of $m \leq n/2$ weaker agreement is prescribed, potentially leading to multiple correct outputs. It is thus seen that Definition 2.1 covers a wide array of voting schemes of common interest.

We will not use Definition 2.1 in its full generality but will deal primarily with exact voting schemes. However, the general definition is crucial for understanding the limitations of our algorithms and for their future extensions. In most practical applications of m-out-of-n voting, the magnitude of m is comparable to n; i.e., $m = \theta(n)$. We will give this important special case, as well as the further restricted case of $n - m = O(1)$ special attention. All logarithms in this paper are base-2.

3. Sequential Threshold Voting

When the input object space is totally ordered, sorting can be used to obtain an $O(n \log n)$ sequential algorithm for weighted threshold voting (assuming, of course, that the relative order of two objects can be established in constant time). This approach applies with equal efficiency to exact and inexact voting. If the input object space is small, say of size δ, then tallying of votes for each

possible object (akin to tallying of votes for a small number of candidates in an election) can be used to obtain an $O(n\delta)$-time algorithm, leading to linear time for any fixed δ. The remaining questions are then: (1) Can one do better than $O(n \log n)$ for th

reshold voting with a large space? (2) What if the object space is unordered, allowing only comparison for equality among objects?

The following algorithm that has been adapted from the unweighted version in [3] provides answers to both questions. It has a time complexity $O(np)$ and requires working space for only p input objects, where $p = \lfloor V/t \rfloor$. Thus, when t is comparable to V in magnitude (as in majority voting), p is a small constant and the algorithm needs linear time and constant working space. On the other hand, for small values of t, the performance deteriorates to quadratic time and linear working space. The latter results are consistent with those of plurality voting under similar conditions.

Since for small t, we have $p = O(V)$, one may think that in this case, the algorithm has time complexity $O(nV)$ which may be much greater than $O(n^2)$. To prove worst-case quadratic time complexity, let the largest input vote be v_{max}. For $t \le v_{max}$, t-out-of-V voting becomes trivial since any input object with vote no less than t is a valid output. The possibility of checking the condition $t \le v_{max}$ in linear time and the fact that for $t > v_{max}$ we have $V/t < n$, prove the worst-case quadratic time complexity. In some such cases, it may be advantageous to use an $O(n^2)$-time plurality voting algorithm to identify a maximum-vote object. The derived vote for this object can then be compared to t to decide on the output.

The special case of $V - t = O(v_{min})$, subsuming unanimity voting with $t = V$, $(n - \varepsilon)$-out-of-n voting, and the like, allows the use of a conceptually simpler algorithm which consists of identifying an arbitrary, constant-size, subset of inputs whose vote total exceeds $V - t$ (perhaps including the maximum-vote input in the set in order to minimize its size), tallying the votes for each member of the set in $O(n)$ time, and doing a final selection in this small set.

3.1. Algorithm (sequential exact threshold voting for large unordered object space): We need working storage space or "slots" for $p = \lfloor V/t \rfloor$ different inputs $object_1, object_2, \ldots, object_p$, with each $object_j$ having an associated vote total $tally_j$.

1. $object_1 := x_1$; $tally_1 := v_1$; $tally_j := 0$ $(2 \le j \le p)$
2. for $i = 2$ to n do {process the remaining $n - 1$ input objects}
3.   if $\exists j$ such that $x_i = object_j$ with $tally_j \ne 0$
4.   then $tally_j := tally_j + v_i$
5.   else if $\exists j$ with $tally_j = 0$ {an empty slot?}
6.   then $object_j = x_i$; $tally_j := v_i$ {save input object in empty slot}
7.   else let $min = tally_k$ be a minimum of all $tally_j$s $(1 \le j \le p)$
8.   if $v_i \le min$
9.   then $tally_j := tally_j - v_i$ $(1 \le j \le p)$
10.   else $object_k := x_i$; $tally_k := v_i$; $tally_j := tally_j - min$ $(1 \le j \le p)$
11.   endif
12.   endif
13.   endif
14. endfor
15. $tally_j := 0$ $(1 \le j \le p)$
16. for $i = 1$ to n do
17.   if $\exists j$ such that $x_i = object_j$
18.   then $tally_j := tally_j + v_i$; if $tally_j \ge t$ then output $object_j$ and stop endif
19.   endif
20. endfor ▮

Following is a textual description of Algorithm 3.1. The first input object and its associated vote are stored in the first slot and all other $tally_j$s are initialized to 0, thus designating the remaining $p - 1$ slots as empty (Line 1). Input objects $x_2, \ldots, x_n$ are then examined in turn (Line 2). The next input object x_i to be considered is compared to the stored objects (Line 3). If $x_i = object_j$ for some j, then $tally_j$ is incremented by v_i (Line 4). If x_i is not equal to any $object_j$ and fewer than p objects have been stored in the p slots, then (x_i, v_i) is stored in an available empty slot (Line 6). If all p slots are occupied, then the minimum vote tally $min = tally_k$ for the stored objects is found (Line 7). If $v_i \le min$, then the new input x_i is discarded and all stored vote tallies are reduced by v_i (Line 9). If $v_i > min$, then all vote tallies are reduced by min and $(x_i, v_i - min)$ replaces one of the objects which is left with 0 vote tally (Line 10). A second pass through the input, comparing each x_i to all $object_j$s, tallying the actual vote for each $object_j$, and stopping as soon as some tally reaches the threshold t, completes the algorithm (Lines 15-20). For proof of correctness see [17].

3.2. Example: Let the n object/vote pairs be denoted as (x_i, v_i), $i = 1, 2, \ldots, n$. Consider 6-way, 8-out-of-15 voting with vote weights 4, 3, 3, 2, 2, 1. Take an instance of the voting problem with inputs $(A, 3)$, $(B, 2)$, $(B, 2)$, $(A, 1)$, $(C, 3)$, $(A, 4)$ in presentation or input order. Since $V = 15$, $t = 8$, and $p = \lfloor 15/8 \rfloor = 1$, a single working storage slot $(object_1, tally_1)$ is required. This slot will hold the values $(A, 3)$, $(A, 1)$, $(B, 1)$, $(-, -)$, $(C, 3)$, and $(A, 1)$ successively as we proceed through the steps of Algorithm 3.1. Therefore, A is a candidate value for the voting result and a second pass through the input will yield its actual vote tally of 8 for comparison with the threshold 8. ▮

Since when only equality comparison is allowed for the input objects the evaluation of the condition on Line 3 of Algorithm 3.1 involves an $O(p)$-time linear search, the overall algorithm time complexity is $O(np)$. With a totally ordered object space, the linear search can be replaced by a $(\log p)$-time operation on a search structure such as a binary tree. This would be beneficial only when p is relatively large. The resulting time complexity is $O(n \log p)$ which would be comparable to $O(n \log n)$ for small weights and thresholds. Note that, as written, the above modifications to the search on Line 3 does not reduce the algorithm complexity in view of the implicit $O(p)$-time loops on Lines 7 and 10. However, these $O(p)$-time loops can be removed by keeping a second version of the list in total vote order and by maintaining a "total vote reduction" variable instead of actually modifying the p votes each time. Hash coding may also be applicable for certain types of input objects and input distributions to reduce the search time to $O(1)$ and the overall time complexity to $O(n)$ on the average.

4. Parallel m-out-of-n Voting

One can parallelize Algorithm 3.1 based on a divide-and-conquer strategy. In this paper, we deal with the unweighted case only (i.e., $v_i = 1$, $V = n$, $t = m$) and assume that n is a power of 2. Implications of removing these assumptions will be discussed briefly in Section 7. The two phases of the algorithm (identifying up to $p = \lfloor n/m \rfloor$ candidates, Lines 1-14, and selecting those candidates whose vote total actually exceeds m, Lines 15-20) will be merged in the following discussion in the sense that once the p candidates have been identified, they carry with them their total vote tally from all inputs. The second pass is thus not needed.

Let us divide the n inputs into two equal subsets $x_1, x_2, \ldots, x_{n/2}$ and $x_{n/2+1}, x_{n/2+2}, \ldots, x_n$. If an object is to have a total of m votes, it must have at least $m/2$ votes in one of the two subsets. Using the parallel threshold voting algorithm recursively on each subset, one can identify p or fewer candidates, along with their associated votes, in each subset. Clearly, if m is even, then $m/2$-out-of-$n/2$ voting leads to no more than $\lfloor (n/2)/(m/2) \rfloor = \lfloor n/m \rfloor = p$ candidates. For odd m, $(m+1)/2$-out-of-$n/2$ voting is performed on the two subsets which again leads to no more than p candidates. The remaining problem then is to merge the two lists of up to p candidates into a single list of p candidates, combining the votes of common elements in the two lists and discarding some of the lower-vote items along the way.

Let the two candidate lists be $object_1, object_2, \ldots, object_p$ and $object'_1, object'_2, \ldots, object'_p$. Assume that the exact vote totals for each list are also provided by the recursively called algorithm. Let these vote totals be $tally_1, tally_2, \ldots, tally_p$ and $tally'_1, tally'_2, \ldots, tally'_p$, respectively. To find the exact votes associated with each $object_i$ in the entire set of inputs, the vote total $tally_i$ for $object_i$ in the first subset must be augmented with corresponding votes from the second subset (and similarly for $object'_i$). Hence, the processors computing the $object_i$ and $object'_i$ entities and their associated vote totals within each subset must exchange the two sets of results and proceed to update the votes. Many algorithm details and their time complexities depend on the particular model of parallel computation assumed. These will be discussed later. Once the two lists of size p along with associated total votes are available, one can sort the combined list of size $2p$ by total votes, remove duplicate elements (that are necessarily adjacent in the sorted list), and keep up to p elements whose total vote tallies are at least m. The above discussion leads to the following high-level description of the parallel m-out-of-n threshold voting algorithm.

4.1. Algorithm (parallel exact m-out-of-n voting for large unordered object space): We need working storage space or "slots" for $2p = 2\lfloor n/m \rfloor$ object-vote pairs $(object_1, tally_1)$, $(object_2, tally_2), \ldots, (object_p, tally_p)$ and $(object'_1, tally'_1)$, $(object'_2, tally'_2), \ldots, (object'_p, tally'_p)$ for each recursive call. The final p candidates and their associated votes will be returned in $(object_1, tally_1)$, $(object_2, tally_2), \ldots, (object_p, tally_p)$.

1. Divide the inputs into two equal-size subsets $S = \{x_1, x_2, \ldots x_{n/2}\}$, $S' = \{x_{n/2+1}, x_{n/2+2}, \ldots x_n\}$
2. Perform $\lfloor (m+1)/2 \rfloor$-out-of-$n/2$ voting in parallel on S and S', with results returned in $(object_1, tally_1), (object_2, tally_2), \ldots, (object_p, tally_p)$ for S and $(object'_1, tally'_1)$, $(object'_2, tally'_2), \ldots, (object'_p, tally'_p)$ for S', where each pair is an object with associated total vote within its subset.
3. Exchange the two sets of results $(object_i, tally_i)$ and $(object'_i, tally'_i)$ from Step 2 such that processors that worked on S have access to the results for S', and vice versa.
4. Update the vote tallies $tally'_1, tally'_2, \ldots, tally'_p$ by adding to each $tally'_i$ the votes of all equal objects in S. Similarly, update $tally_1, tally_2, \ldots, tally_p$ by incorporating the votes from S'.
5. Sort each list in descending order of the vote totals from all inputs ($tally_1, tally_2, \ldots, tally_p$ or $tally'_1, tally'_2, \ldots, tally'_p$).
6. Merge the object-vote pairs $(object'_1, tally'_1)$, $(object'_2, tally'_2), \ldots, (object'_p, tally'_p)$ into the list $(object_1, tally_1)$, $(object_2, tally_2), \ldots, (object_p, tally_p)$, removing duplicate entries if any, until all p slots are occupied. Ignore any remaining object. ∎

The following example should clarify the algorithm.

4.2. Example: Consider 8-way, 3-out-of-8 voting with equal vote weights. Take an instance of the

voting problem with inputs A, A, D, D, B, C, A, B in presentation or input order. Since $n = 8$, $m = 3$, and $p = \lfloor 8/3 \rfloor = 2$, four working storage slots, $(object_1, tally_1)$, $(object_2, tally_2)$ for $S = \{A, A, D, D\}$ and $(object'_1, tally'_1)$, $(object'_2, tally'_2)$ for $S' = \{B, C, A, B\}$, are initially required. These slots will hold $(A, 2)$, $(D, 2)$ and $(B, 2)$, $(-, 0)$ following the recursive calls in Step 2, indicating that A, D, and B are possible candidates for the voting result in view of getting at least 2 votes in one of the halves. Exchanging the two lists of size 2 and adding in the corresponding votes in the other half leads to the updated lists $(B, 2)$, $(-, 0)$ and $(A, 3)$, $(D, 2)$. Merging the two lists yields $(A, 3)$, $(-, 0)$, which indicates that A with 3 votes is the unique answer. The pair $(-, 0)$ stands for a dummy or null entry in an intermediate or final candidate list. ∎

In the special case of majority voting, the above algorithm can be simplified and converted to a variant of a previously published parallel majority voting algorithm [10]. In majority voting, i.e. for $m = n/2 + 1$, the recursive calls in Step 2 of Algorithm 4.1 lead to a single candidate each, say $(object_1, tally_1)$ and $(object'_1, tally'_1)$. The merging phase (replacing Steps 3-6 in Algorithm 4.1) is now quite simple. If $object_1 = object'_1$, then the merge result is easily obtained as $(object_1, tally_1 + tally'_1)$. Otherwise, the object with the larger vote is the only possible candidate for majority (equal votes imply that there is no majority). Hence, the one candidate for continuing the algorithm at the next level is identified with a single comparison and there is no need to tally the votes from the other half for each of the two objects.

To see this, let $tally_1 = n/4 + a$ and $tally'_1 = n/4 + b$, with $a \geq b > 0$. The maximum vote that $object'_1$ (the object with the smaller vote) can get in the first half is $n/4 - a$, since there are a total of $n/2$ objects and $object_1$ has $n/4 + a$ votes. Thus the total vote associated with $object'_1$ can be no more than $n/4 + b + n/4 - a = n/2 - (a - b)$ which is less than majority. Clearly, the above argument and the resulting simplification also apply to super-majority threshold voting schemes; i.e., the case of $m > n/2 + 1$.

5. Analysis for $m = \theta(n)$

In this section, we first present a general analysis of Algorithm 4.1 that includes among its parameters the time to perform various key "building-block" operations on the parallel architecture of interest. We then derive exact complexities for several well-known parallel architectures as examples. In what follows, we will assume $p = \lfloor n/m \rfloor$ to be a (small) constant. This is consistent with most uses of threshold voting in practice. The condition $m = \theta(n)$ allows us to simplify the analysis by assuming that p objects can be stored and/or manipulated in the local memory associated with a single processor of constant complexity.

Let $T(n, p)$ be the time needed for m-out-of-n voting for $m = \theta(n)$. Step 1 of Algorithm 4.1 takes constant time since it only involves designating each input object as a member of one or the other subset. Step 2 takes $T(n/2, p)$ time, assuming that each half of the system has the same connectivity or architecture as the entire system. This is the case in virtually all parallel architectures of practical interest. Let $T_E(p)$ be the time needed to exchange p values between the two halves in Step 3. Similarly, let $T_A(n/2, p)$, $T_S(p)$, and $T_M(p, p)$ be the vote accumulation, sorting, and merging times of Steps 4 through 6, respectively. With these assumptions, the worst-case running time of the algorithm is defined by the recurrence:

$$T(n, p) = T(n/2, p) + T_E(p) + T_A(n/2, p) + T_S(p) + T_M(p, p)$$

The following examples illustrate the actual time complexity of Algorithm 4.1 for several well-known parallel architectures. In all cases, multiple processors are assumed to operate in SIMD mode (under control of a single instruction stream).

5.1. Example: Consider the PRAM model of parallel computation with concurrent reads allowed (CREW). For this model, T_E is a constant as the exchange involves reading a pointer to the corresponding p-element array by $n/2$ processors. On the other hand, T_A consists of p fan-in or semigroup computations (summations) and takes $p \log(n/2)$ time. $T_S = O(p \log p)$, assuming that a single processor does the sorting in each list. Finally, $T_M = O(p)$. Hence the recurrence becomes $T(n) = T(n/2) + O(p \log n) = O(\log^2 n)$, given the assumption that p is a small constant. For the EREW (exclusive read) PRAM model, broadcasting of p values to $n/2$ processors in the other subset takes $O(p \log n)$ time, so the overall complexity is asymptotically the same. ∎

5.2. Example: Consider the hypercube model of parallel computation and assume that the results of the two halves are held in neighboring processors. For this model, T_E is p or $2p$, depending on simplex or full duplex exchange of values between the neighbors. T_A consists of p broadcast operations, each followed by a fan-in computation (summation) and takes $O(p + \log n)$ time, using standard pipelining techniques. $T_S = O(p \log p)$, assuming that a single processor does the sorting in each list. Finally, $T_M = O(p)$. Hence, in this case, the recurrence for time complexity becomes $T(n) = T(n/2) + O(\log n) = O(\log^2 n)$, given the assumption that p is a small constant. ∎

The $O(\log^2 n)$ complexity obtained in Example 5.2 for hypercube holds for a wide variety of other hypercubic (hypercube-derivative) and other scalable constant-degree networks such as butterflies, cube-connected cycles, shuffle-exchange networks [11], and periodically regular chordal rings [19]. It also holds for any architecture that

can emulate hypercube algorithms incorporating single-dimension communication (a special case of single-port communication, whereby all nodes are constrained to communicate along the same dimension in each step) with constant slowdown. Examples include star and star-connected cycles networks [9], hierarchical cubic networks [6], as well as wide classes of recursively constructed networks proposed recently [23], [24].

5.3. Example: Consider the k-D mesh model of parallel computation and assume that the results of the two halves are held in neighboring processors. For this model, T_E is again $O(p)$. T_A consists of p broadcasts, each followed by a fan-in computation (summation) and thus takes $O(p + n^{1/k})$ time, using standard pipelining techniques. $T_S = O(p \log p)$, assuming that a single processor does the sorting in each list. Finally, $T_M = O(p)$. Hence the recurrence becomes $T(n) = T(n/2) + O(n^{1/k}) = O(n^{1/k})$, given the assumption that p is a small constant. ∎

Example 5.3 shows that the algorithm is asymptotically optimal for the k-D mesh model. However, the time complexities of Examples 5.1 and 5.2 are suboptimal. As of this writing, a more clever way of organizing the computations in order to reduce this complexity is not known

Analysis of the algorithm in the special case of majority or super-majority voting, as discussed at the end of Section 4, is quite simple and leads to the recurrence $T'(n) = T'(n/2) + O(1) = O(\log n)$ for identifying the final candidate on PRAM and hypercubic models and to $T'(n) = T'(n/2) + O(n^{1/k}) = O(n^{1/k})$ for the k-dimensional mesh model. To this, one must add the time $T_V(n)$ needed for the final verification of majority status among the n inputs, leading to $T(n) = T'(n) + T_V(n)$. Since the final verification of majority status can also be done in $O(\log n)$ or $O(n^{1/k})$ time on the PRAM/hypercube and k-dimensional mesh models, respectively, an efficient asymptotically optimal algorithm results in each case.

One can consider the further restricted case of super-majority where $n - m = O(1)$. This is when $(n - \varepsilon)$-out-of-n voting is required for a (small) constant ε. The only additional simplification in this case occurs for some CRCW PRAM submodels. For example, when multiple writes are allowed in case a common value is written or when upon multiple writes of single-bit values, the logical OR (the maximum) of the values is stored, one can proceed as follows. An $(\varepsilon + 1)$-element list of different inputs is constructed iteratively; start with one element in the list and, in each step, have one of the processors holding a value different from the ones on the list add its value to the list. Next, all n processors compare their values to those on this list and in case of disagreement, write a 1 into the corresponding location of an $(\varepsilon + 1)$-element "disagreement" array which is initialized to all 0s. An item can have $n - \varepsilon$ or more votes only if no disagreement is registered for it. Therefore, after the above constant-time steps, the "disagreement" array points to the correct voting result, if any, or indicates that there is no input element which has at least $n - \varepsilon$ votes.

6. Analysis for $m = o(n)$

When m is much smaller than n, the parameter $p = \lfloor n/m \rfloor$ is no longer a constant and it is unreasonable to base the complexity analysis on the assumption that a single processor can hold and efficiently sort/merge lists of size p. More importantly, if the input object space is large and unordered, only direct comparison can be used to establish the equality of objects, leading to $\Omega(p)$ time complexity for any vote accumulation scheme and $\Omega(p \log n)$ time overall. With $m = o(n)$, the above can be as large as $\Omega(n \log n)$. We thus assume a totally ordered input object space that would allow us to use sorting to simplify the vote accumulation phase. The recurrence

$$T(n, p) = T(n/2, p) + T_E(p) + T_A(n/2, p) + T_S(p) + T_M(p, p)$$

derived in Section 5 is still valid and can be used as a starting point for our analysis. Since the storage and/or transformation of the required lists of size p must now be distributed across multiple processors, the distribution of data/processing becomes critical in minimizing the time complexity. Thus, we revisit each of the three examples of Section 5, with the aim of specifying efficient distribution schemes to minimize the overall complexity. As before, a SIMD mode of operation is assumed.

6.1. Example: Consider the PRAM model, with or without concurrent reads (CREW/EREW). The differences with Example 5.1 occur in the vote accumulation, sorting, and merging steps. For efficiency, we base the vote accumulation step on sorting rather than on multiple semigroup computations. The sorting of x items using x processors takes $O(\log x)$ time. Thus, both the $n/2$-element subset and the p-element candidate list can be sorted in $O(\log n)$ time. Vote tallying can then be accomplished by merging the $n/2$-element list of inputs into the p-element list of candidates in $O(\log n)$ steps. This observation, combined with the fact that p-element lists can be sorted/merged in $O(\log p)$ time using p processors, leads to the conclusion that the time complexity of the algorithm remains $O(\log^2 n)$. In fact, the algorithm can be made even more efficient by sorting the input list initially. This would lead to all lists and sublists being automatically obtained in sorted order in intermediate steps. So, the need for multiple sorts is eliminated. However, since vote accumulation involves the $(\log n)$-time merging of two lists of sizes $n/2$ and p using $n/2$ processors, the asymptotic time complexity is not affected. ∎

6.2. Example: Consider the hypercube model of parallel computation and assume that the results of the two halves are held in directly linked $(\log p)$-dimensional subcubes of the two $\log(n-1)$-cubes. In this case, T_E becomes a constant term (1 or 2). Vote accumulation involves the sorting and merging of $n/2$ elements on a $\log(n-1)$-cube into p elements on one of its $(\log p)$-dimensional subcubes. This is easily accomplished in $O(\log n)$ time using known sort/merge algorithms. The original sorting and merging steps based on vote totals take $O(\log p)$ time within the respective subcubes. Hence, given that the $O(\log n)$ term dominates the $O(\log p)$ terms, the recurrence for time complexity remains $T(n) = T(n/2) + O(\log n) = O(\log^2 n)$. ∎

As before, the $O(\log^2 n)$ complexity obtained in Example 6.2 for hypercube holds for a wide variety of fixed-degree and hierarchical networks that were enumerated following Example 5.2.

6.3. Example: Consider the k-D mesh model of parallel computation and assume that the results from the two halves are held in adjacent k-D submeshes of size $p^{1/k}$. For this model, T_E is $O(p^{1/k})$. The sorting and merging operations required for vote accumulation take $O(n^{1/k})$ time. The original sorting/merging steps based on vote totals require $O(p^{1/k})$ time within the respective submeshes. Hence the recurrence for time complexity remains $T(n) = T(n/2) + O(n^{1/k}) = O(n^{1/k})$, given that the $O(n^{1/k})$ term dominates the $O(p^{1/k})$ terms. ∎

Again, we see that the algorithm is asymptotically optimal for the k-D mesh model and suboptimal for the other models examined.

It is unlikely that more efficient algorithms can be designed when the voting threshold m is further restricted, to $m = O(1)$ or $m = O(\log n)$, say. But we have no proof for this.

7. Some Extensions

When n is not a power of 2, a simple modification to the algorithm can be used that does not affect the asymptotic time complexities discussed thus far. The modification consists of padding the list of n elements with $2^{\lfloor \log n \rfloor + 1} - n$ "wild" objects that would match any object in equality comparisons. This would inflate all votes by $2^{\lfloor \log n \rfloor + 1} - n$, and thus has no effect on relative order of total votes. Subtracting $2^{\lfloor \log n \rfloor + 1} - n$ from the final vote tallies or adding it to the threshold t corrects the bias. Alternatively, one could use weighted threshold voting, as discussed below, with $2^{\lfloor \log n \rfloor + 1} - n$ additional arbitrary objects whose votes are set to 0, but this is bound to be less efficient.

So far we have based our analyses on the assumption that we have n processors to solve our n-input voting problem. When n is large, it would be more reasonable to assume that the number q of processors satisfies $q < n$. In this case, we can use the same algorithm by requiring that each of the q processors emulate n/q processors in the original solution with n processors. For the PRAM version of the algorithm, this will result in a slowdown factor of n/q, leading to the overall time complexity $O((n/q)\log^2 n)$. In the case of hypercube, each processor must emulate an (n/q)-node subcube, leading to the same slowdown as for the PRAM. Finally, for a k-D mesh, each processor must emulate a k-D submesh of dimension $(n/q)^{1/k}$. This again leads to the same optimal slowdown.

Weighted voting would involve the following modifications to the algorithm and its analysis. Ideally, one would like to replace Step 1 of Algorithm 4.1 with a partitioning of the set of inputs into roughly equal-weight subsets. This would lead to only minor changes in the remaining steps, as the number of candidates in each subset will be roughly equal to p. However, for arbitrary weights, such a partitioning is a special case of the subset sum problem which is known to be NP-complete [4]. Thus, we opt for partitioning into equal-size subsets as before. The total votes associated with the two subsets will be roughly equal in a probabilistic sense, thus making our previous analyses valid for expected (rather than worst-case) time complexity. Thus, the PRAM/hypercube versions of our algorithm run in expected $O(\log^2 n)$ time in the same sense that sequential quicksort runs in $O(n \log n)$ time on the average.

One way to derive the worst-case time complexity of the weighted voting version of our algorithm is to take the minimum weight to be $v_{min} = 1$ and assume a maximum weight of v_{max}. Then, the number of candidates having at least $t/2$ votes in each subset of $n/2$ elements is no more than $\lfloor (nv_{max}/2)/(t/2) \rfloor \le pv_{max}$. Thus, the preceding asymptotic analyses still apply if v_{max} is a constant.

8. Concluding Remarks

We have presented a parallel algorithm for m-out-of-n threshold voting and analyzed its complexity for various models of parallel computation. The algorithm was shown to be optimal for k-D mesh-connected parallel computers and reasonably efficient on the PRAM, hypercube, some fixed-degree derivative networks, and a wide array of hierarchical architectures. In the special case of majority voting, it was shown that the algorithm can be modified to run in optimal time on all of the above models. The key to this simplification in the case of majority voting is that merging and selection of candidates in each recursive step does not require the tallying of votes from the other partition; a simple comparison of votes tells us which of the two candidates may have majority, with the actual tallying of votes deferred until a single candidate has been identified at the very end. Thus, the vote-tallying overhead is paid only once.

When the input object space is totally ordered, asymptotically optimal algorithms can be constructed if the objects are first sorted using any optimal sorting algorithm. Once the input objects are sorted, vote tallying can be done by a partitioned parallel prefix computation, and the outputs can be selected by a parallel comparison of vote tallies with the threshold t. However, even in such cases, our algorithm may be faster in view of potentially smaller constants.

Work is expected to continue on refining the algorithm and its analysis and on evaluating its efficiency on various other models of parallel computation. Of particular interest is to show how tallying of votes in each recursive step can be avoided for general threshold voting or to prove that the $O(\log^2 n)$ complexity obtained for the PRAM, hypercube, and related hypercubic networks is in fact the best possible (a lower bound). Extension of the algorithm to more efficient weighted voting schemes, particularly when v_{max}/v_{min} is not small, constitutes another direction for further research. Finally, we have only dealt with threshold voting in this paper. Even though parallel complexity results have been previously derived for a variety of voting schemes [15], efficient parallel algorithms for these cases remain to be developed.

References

[1] R.S. Boyer and J.S. Moore, "MJRTY – a fast majority vote algorithm", in *Automated Reasoning: Essays in Honor of Woody Bledsoe*, Ed. R. Boyer, 1991; Kluwer.

[2] D.R.K. Brownrigg, "The weighted median filter", *Communications of the ACM*, vol 27, no 8, pp 807-818, 1984 Aug.

[3] D. Campbell and T. McNeill, "Finding a majority when sorting is not available", *The Computer Journal*, vol 34, no 2, p. 186, 1991 Apr.

[4] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, pp 951-953, 1990; McGraw-Hill.

[5] D. Dolev, L. Lamport, M. Pease, and R. Shostak, "The Byzantine generals", in *Concurrency Control and Reliability in Distributed Systems*, Ed. by B.K. Bhargava, pp 348-369, 1987; Van Nostrand Reinhold.

[6] K. Ghose and R. Desai, "Hierarchical cubic networks", *IEEE Trans. Parallel Distributed Systems*, vol 6, no 4, pp 427-435, 1995 Apr.

[7] D. Gries, "A hands-in-the-pocket presentation of a k-majority vote algorithm", in *Formal Development of Programs and Proofs*, Ed. by E.W. Dijkstra, pp 43-45, 1990; Addison-Wesley.

[8] S. Iyengar, S. Sitharama, R. Kashyap, and R. Madan (Guest Eds.), Special section on distributed sensor networks, *IEEE Trans. Systems, Man, and Cybernetics*, vol 21, no 5, pp 1027-1031, 1991 Sep/Oct.

[9] S. Latifi, M. Azevedo, and N. Bagherzadeh, "The star-connected cycles: a fixed-degree network for parallel processing", *Proc. Int'l Conf. Parallel Processing*, vol I, pp 91-95, 1993 Aug.

[10] C.-L. Lei and H.-T. Liaw, "Efficient parallel algorithms for finding the majority element", *J. Information Science & Engineering*, vol 9, pp 319-334, 1993.

[11] F.T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, 1994; Morgan Kaufmann.

[12] P.R. Lorczak, A.K. Caglayan, and D.E. Eckhardt, "A theoretical investigation of generalized voters for redundant systems", *Proc. Int'l Symp. Fault-Tolerant Computing* (Chicago), pp 444-451, 1989 Jun.

[13] J. Misra and D. Gries, "Finding repeated elements", *Science of Computer Programming*, vol 2, 1982, pp 143-152. See also a related correspondence item in *The Computer J.*, vol 35, no 3, p 298, 1992 Jun.

[14] B. Parhami, "Voting networks," *IEEE Trans. Reliability*, vol 40, pp 380-394, 1991 Aug.

[15] B. Parhami, "The parallel complexity of weighted voting", *Proc. Int'l Symp. Parallel and Distributed Computing and Systems* (Washington, DC), pp 382-385, 1991 Oct.

[16] B. Parhami, "Optimal algorithms for exact, inexact, and approval voting", *Proc. Int'l Symp. Fault-Tolerant Computing* (Boston), pp 444-451, 1992 Jun.

[17] B. Parhami, "Threshold voting is fundamentally simpler than plurality voting", *Int'l J. Reliability, Quality and Safety Engineering*, vol 1, no 1, pp 95-102, 1994.

[18] B. Parhami, "Voting algorithms", *IEEE Trans. Reliability*, vol 43, no 4, pp 617-629, 1994 Dec.

[19] B. Parhami, "Periodically regular chordal ring networks for massively parallel architectures", *Proc. 5th Symp. Frontiers of Massively Parallel Computation* (McLean, VA), pp. 315-322, 1995 Feb.

[20] B. Parhami, "Multi-sensor data fusion and reliable multi-channel computation: unifying concepts and techniques", *Proc. Asilomar Conf. Signals, Systems, and Computers* (Pacific Grove, CA), 1995 Oct.

[21] B. Parhami, "A taxonomy of voting schemes for data fusion and dependable computation", *Reliability Engineering and System Safety*, to appear.

[22] S.Y.H. Su, M. Cutler, and M. Wang, "Self-diagnosis of failures in VLSI tree array processors", *IEEE Trans. Computers*, vol 40, no 11, pp 1252-1257, 1991 Nov.

[23] C.-H. Yeh and B. Parhami, "Swapped networks: unifying the architectures and algorithms of a wide class of hierarchical parallel processors", *Proc. Int'l Conf. Parallel & Distributed Systems* (Tokyo), 1996 Jun.

[24] C.-H. Yeh and B. Parhami, "Unified formulation of a wide class of scalable interconnection networks based on recursive graphs", *Proc. 11th Int'l Conf. Systems Engineering* (Las Vegas, NV), 1996 Jul.