Parallel Threshold Voting

BEHROOZ PARHAMI

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

Voting on large collections of input objects is becoming increasingly important in data fusion, signal and image processing applications, learning algorithms, 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. Our discussion centers on unweighted threshold (m-out-of-n) voting. However, we observe that under certain conditions, the results can be extended to efficient weighted threshold voting. We show how a 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 n-processor PRAM and hypercubic computers and $O(k^2 n^{1/k})$ time complexity on a k-dimensional mesh with n processors. We also analyze the time complexity of the algorithm for $m = o(n)$ and its special case of $m = \theta(1)$.

Received August 14, 1995; revised June 28, 1996

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 [1]. 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 components with diverse data, or various hybrid combinations of hardware/program/data redundancy and/or diversity [2].

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 collection of sensors [1, 3, 4], image processing filters which smooth digital pictures by voting on neighborhoods [5], enhancement of learning algorithms [6], implementation of cellular automata and neural networks [7], and certain configuration control, bookkeeping and diagnostic functions in parallel and distributed systems [8–10].

Threshold voting, in particular, is more likely to be useful for such large-scale applications in view of its inherent simplicity compared with other voting schemes [11]. In its unweighted (or m-out-of-n) form, threshold voting is used to identify a certain required minimum level of agreement among multiple information sources. All values, from 2 to n, are meaningful for the threshold level m. Some of these values even lead to applications beyond those discussed in the preceding paragraph. The case of $m = 2$ corresponds to detecting the presence of repeated elements in a set which finds applications in the study of chemical structures. The case $m = n - \epsilon$, for some small fixed ϵ, can be viewed as near-unanimity voting for use when no more than ϵ faults/disagreements are expected.

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 [12–14], sequential threshold voting [15] and parallel majority voting [16]. 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 [11, 17–19]. In this paper, we generalize and extend the above to efficient parallel m-out-of-n threshold voting algorithms. We also note that our results can be extended to certain weighted threshold voting schemes without serious loss in algorithm speed. Parallel algorithms for general weighted threshold voting remain to be developed.

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. In Section 4, we show how the unweighted m-out-of-n version of this algorithm can be parallelized based on a 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 threshold voting. We conclude, in Section 8, with

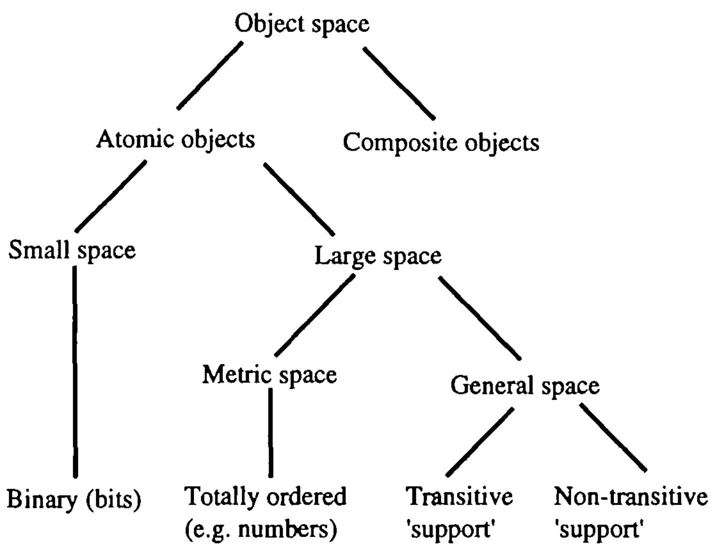

FIGURE 1. Voting schemes classified according to the object space size and structure.

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 [20] or voting routines) and based on the size and structure of the input object space (see Figure 1). A voting algorithm [19] 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.

The input objects of a voting routine may be atomic or composite (Figure 1). Composite objects, consisting of 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. 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 is crucial. In particular, the special case of a totally ordered object space leads to simple and efficient algorithms. 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 '$object_1$ supports $object_2$' and '$object_2$ supports $object_3$' together imply that '$object_1$ supports $object_3$').

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

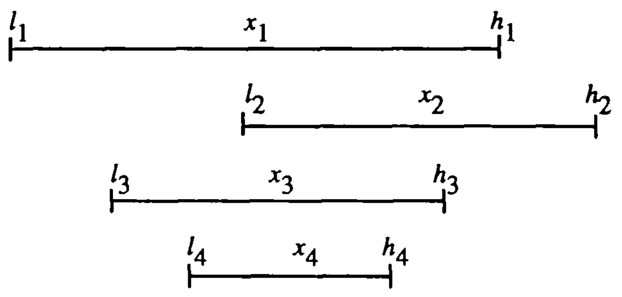

FIGURE 2. Voting with composite data objects represented as intervals on the real line.

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, then majority consensus can still be reached.

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

DEFINITION 2.1. *(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.*

The term 'supported by', which accounts for the generality of Definition 2.1, can be defined in several ways, leading to different threshold 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). Various constraints on the output vote w further define a number of voting subschemes within the above voting classes; 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 will be assumed.

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. We also briefly deal with the case of $m = o(n)$ for completeness. 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: (i) Can one do better than $O(n \log n)$ for threshold voting with a large space? (ii) What if the object space is unordered, allowing only comparison for equality among objects?

The following algorithm, adapted from the unweighted version in [15], 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 $t \leq v_{\max}$, 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 $t \leq v_{\max}$ 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 [19] to identify a maximum-vote object. The derived vote for this object can then be compared with t to decide on the output.

The special case of $V - t = O(v_{\min})$, subsuming unanimity voting with $t = V$, $(n - \epsilon)$-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.

ALGORITHM 1. (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 $tally_j$. The algorithm is shown in Figure 3.

EXAMPLE 1. 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 threshold 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 which will hold the values $(A, 3), (A, 1), (B, 1), (-, -), (C, 3)$ and $(A, 1)$ successively as we proceed through the steps of Algorithm 1. Therefore, A is a candidate for the voting result and a second pass through the input yields its actual vote tally of 8 for comparison with the threshold.

Since when only equality comparison is allowed for the input objects the evaluation of the condition on line 3 of Algorithm 1 (Figure 3) 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 $O(n \log p)$ or $O(n \log n)$ for small weights and thresholds. Note that, as written, the above modifications to the search on line 3 do 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

```
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
```

FIGURE 3. The sequential threshold voting algorithm (Algorithm 1).

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 complexity to $O(n)$ on the average.

4. PARALLEL m-OUT-OF-n VOTING

One can parallelize Algorithm 1 based on a divide-and-conquer strategy. We deal primarily with the unweighted case (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 will 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.

ALGORITHM 2. (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 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 $(object_1,$

$tally_1$), ($object_2$, $tally_2$), ..., ($object_p$, $tally_p$) for S and ($object'_1$, $tally'_1$), ($object'_2$, $tally'_2$), ..., ($object'_p$, $tally'_p$) 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 ($object'_1$, $tally'_1$), ($object'_2$, $tally'_2$), ..., ($object'_p$, $tally'_p$) into the list ($object_1$, $tally_1$), ($object_2$, $tally_2$), ..., ($object_p$, $tally_p$), removing duplicate entries if any, until all p slots are occupied or the vote total drops below m. Ignore any remaining object.

The following example should clarify the algorithm.

EXAMPLE 2. 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. With $n = 8, m = 3$ 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 two 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 three votes is the unique answer. The pair $(-, 0)$ stands for a dummy/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 [16]. In majority voting, i.e. for $m = n/2 + 1$, the recursive calls in Step 2 of Algorithm 2 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 2) 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 $m > n/2 + 1$. Since we do not tally the total votes in each recursion step, the final candidate may in fact not possess majority support (this can only happen if there is no majority). Verifying the majority status of the final candidate is easy and requires only a single broadcast step and one fan-in (summation of votes) step.

Algorithm 2 was described as 2-way divide-and-conquer for simplicity and clarity of exposition. It can easily be extended to 2^k-way divide-and-conquer where in each of the 2^k subproblems, $\lfloor (m + 2^k - 1)/2^k \rfloor$-out-of-$n/2^k$ threshold voting is performed and k phases of pairwise combining of p-element lists is required to find the overall list of p candidates. We will see in subsequent sections that multi-way divide-and-conquer is more appropriate in some cases.

5. ANALYSIS FOR $m = \theta(n)$

In this section, we first present a general analysis of Algorithm 2 that includes among its parameters the time to perform various key 'building-block' operations on the parallel architecture of interest. We then derive asymptotic 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 required for m-out-of-n voting for $m = \theta(n)$. Step 1 of Algorithm 2 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 many 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 asymptotic time complexity of Algorithm 2 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).

EXAMPLE 3. Consider the PRAM model of parallel computation with concurrent reads allowed (CREW). Consider first the 2-way divide-and-conquer strategy. 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 $T(n, p) = T(n/2, p) + O(p\log n) = O(\log^2 n)$, 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. Using 2^k-way divide-and-conquer is not of help here since it reduces the number of recursive steps by a factor of k but increases the combining work in each recursive step by roughly the same factor.

EXAMPLE 4. Consider the hypercube model of parallel computation and assume that the results of each half are available in all participating 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 fan-in computations (summations), one for each of the candidates coming from the other half, and takes $O(p + \log n)$ time using standard pipelining techniques. $T_S = O(p\log p)$, with each processor doing the sorting of its p-element list. Finally, $T_M = O(p)$. Hence, in this case, the recurrence for time complexity becomes $T(n, p) = T(n/2, p) + O(\log n) = O(\log^2 n)$, given the assumption that p is a small constant. As in Example 3, 2^k-way divide-and-conquer does not help here.

The $O(\log^2 n)$ complexity obtained in Example 4 for hypercube holds for a wide variety of other hypercubic (hypercube-derivative) and many scalable constant-degree networks such as butterflies, cube-connected cycles, shuffle-exchange networks [21], and some periodically regular chordal rings [22]. 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 slow down. Examples include star and star-connected cycles networks [23], hierarchical cubic networks [24], as well as wide classes of recursively constructed networks proposed recently [25, 26].

EXAMPLE 5. Consider the k-D mesh model of parallel computation. For this model, it is more appropriate to use a 2^k-way divide-and-conquer strategy (e.g. 4-way in a square 2-D mesh) which leads to meshes with equal sides for the subproblems. Assume, as in Example 4, that the results of each partition are available in all participating processors. The combining step is done in k phases, one for each mesh dimension. For each phase, the exchange time is $T_E = O(n^{1/k})$ and the vote accumulation takes $T_A = O(kn^{1/k})$ steps. Sorting and merging require $O(p\log p)$ and $O(p)$ steps, respectively. Hence the recurrence for time complexity becomes $T(n, p) = T(n/2^k, p) + O(k^2 n^{1/k}) = O(k^2 n^{1/k})$, given the assumption that p is a small constant.

The above examples show that the algorithm is asymptotically suboptimal for all three models considered. The hypercube and k-D mesh models have diameter-based lower bounds of $\log n$ and $k(n^{1/k} - 1)$, respectively. As of this writing, it is not known whether a more clever way of organizing the computations can reduce these complexities to the optimal $O(\log n)$ and $O(kn^{1/k})$.

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 of parallel computation and to $T'(n) = T'(n/2^k) + O(kn^{1/k}) = O(kn^{1/k})$ 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 $T(n, 1) = T'(n) + T_V(n)$. Since the final verification of majority status can also be done in $O(\log n)$ or $O(kn^{1/k})$ time on the PRAM/hypercube and k-dimensional mesh models, respectively, an efficient asymptotically optimal algorithm results in each case.

Consider the special case of near unanimity, a limiting case of super-majority with $n - m = O(1)$; i.e. when $(n - \epsilon)$-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 (maximum) of the values is stored, one can proceed as follows. An $(\epsilon + 1)$-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 $(\epsilon + 1)$-element 'disagreement' array which is initialized to all 0s. An item can have $n - \epsilon$ or more votes only if no disagreement is registered for it. Therefore, after the above constant-time steps, the 'disagreement' array points to the voting result, if any, or indicates that there is no input element which has at least $n - \epsilon$ 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 $\Omega(n\log n)$. The recurrence

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

introduced 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 items and processing functions becomes critical in minimizing the time complexity. Thus, we revisit each of the three examples of Section 5, with the aim of specifying suitable data distribution and processing schemes to minimize the overall complexity. As before, a SIMD mode of operation is assumed.

EXAMPLE 6. Consider the PRAM model, with or without concurrent reads (CREW/EREW). The differences with Example 3 occur in the vote accumulation, sorting and merging steps. For vote accumulation, $n/(2p)$ processors can be assigned to tally the votes for each of the $2p$ candidates within its respective $n/2$-element subset, leading to $O(p)$ time steps. Sorting of x items using x processors takes $O(\log x)$ time on the PRAM. Thus, the sort and merge times for vote combining are sublinear in p and can be ignored in our asymptotic analysis. The above lead to $T(n, p) = T(n/2, p) + O(p) = O(p \log n)$. With $p = o(n/\log n)$ or equivalently $m = \omega(\log n)$, the time complexity of parallel threshold voting on PRAM becomes $o(n)$ which is still asymptotically better than that of plurality voting which requires $\Omega(n^2)$ comparisons [20] and thus $\Omega(n)$ time in any n-processor parallel implementation.

EXAMPLE 7. Consider the hypercube model of parallel computation and assume that the list of candidates for the two halves are held in neighboring ($\log p$)-dimensional subcubes of the two $\log(n-1)$-cubes. In this case, T_E becomes a constant term (1 or 2). Vote accumulation involves p broadcast and fan-in computations in the $\log(n-1)$-cubes, needing $O(p+\log n)$ time with pipelining. As in Example 6, we can ignore the sort/merge time which is sublinear in p. Hence, the recurrence for time complexity yields $T(n, p) = T(n/2, p) + O(p+\log n) = O(p \log n + \log^2 n)$. The algorithm complexity in this case ranges from the fairly efficient $O(\log^2 n)$ for $m = \Omega(n/\log n)$ to the less desirable $O(n \log n)$ for $m = O(1)$. It is instructive to carry out the analysis for 2^k-way divide-and-conquer for this example. Once the 2^k problems of size $n/2^k$ are solved, k combining phases are needed with the jth phase requiring $p+\log(n/2^{k-j+1})$ time steps corresponding to data exchange and vote accumulation in neighboring hypercubes of size $n/2^k, n/2^{k-1}, \ldots, n/2$. The recurrence then becomes $T(n, p) = T(n/2^k, p) + k(p + \log n - (k+1)/2)$. Hence, the asymptotic time complexity of the algorithm does not change, given that $k < \log n$.

Again, the $O(p \log n + \log^2 n)$ complexity obtained in Example 7 for hypercube holds for a wide variety of fixed-degree and hierarchical networks that were enumerated following Example 4.

EXAMPLE 8. Consider the k-D mesh model of parallel computation with a 2^k-way divide-and-conquer strategy, as in Example 5, and assume that the results from the 2^k subproblems are held in adjacent k-D submeshes with sides of length $p^{1/k}$. For this model, T_E is $O(p^{1/k})$, since the exchange involves shifting of values along one of the dimensions. The k sort/merge phases required for vote accumulation take $O(kp + k^2 n^{1/k})$ time, since each phase involves p broadcast and fan-in operations with pipelining. The sort and merge steps by vote totals require $O(kp^{1/k})$ time within the respective submeshes. Therefore, the recurrence for time complexity changes to $T(n, p) = T(n/2^k, p) + O(kp + k^2 n^{1/k}) = O(kp \log n + k^2 n^{1/k})$.

Again, we see that the algorithm is asymptotically suboptimal in each of the three cases examined. It is unlikely that more efficient algorithms can be devised when the voting threshold m is further restricted; to $m = \theta(1)$ or $m = \Omega(\log n)$, for example. But we have no proof for this assertion. We note that with $m = \theta(1)$, the time complexity of the algorithm becomes $O(n \log n)$ for Examples 6 and 7 and $O(kn \log n)$ in Example 8. The $O(n \log n)$ complexity is a factor of $\log n$ below the best possible performance in view of the $O(n^2)$ operations required to compare all object pairs. The lower performance of higher-dimensional meshes is not surprising in view of the fact that their corresponding 2^k-way divide and conquer strategy involves a larger amount of data exchange in the combining phase. Again, we do not know whether these are inherent to the problem or are artifacts of our specific algorithm.

For the sake of generality, our analyses thus far were based on an unordered object space where inequality of $object_1$ and $object_2$ does not provide any information about their relationship to $object_3$ (general space, with transitive 'support', in Figure 1). In the case of a totally ordered object space, sorting can be used to obtain highly efficient algorithms for exact (and some inexact) threshold voting schemes. We illustrate this through an example.

EXAMPLE 9. Consider the voting problem of Example 2 with inputs A, A, D, D, B, C, A, B in presentation or input order. Sorting the inputs in lexicographic order yields A, A, A, B, B, C, D, D, with all equal inputs occupying consecutive positions in the resulting sequence. A partition vector $1, 1, 1^t, 1, 1^t, 1^t, 1, 1^t$ is formed to indicate the span of each distinct value, with the tag 't' indicating the last element in each run. Now, a partitioned parallel prefix sum computation, which corresponds to the operator '$+^t$' defined in terms of standard addition as $x +^t y = x + y$, $x +^t y^t = (x + y)^t$, $x^t +^t y = y$, and $x^t +^t y^t = y^t$, is performed on the partition vector. This results in the accumulation of all votes in the tagged elements of the partition vector, making it easy to complete the threshold voting algorithm or to devise a plurality voting algorithm to select one of the inputs with maximum vote.

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 $2^{\lfloor \log n \rfloor + 1} - n$ special objects that would not match any object (including another special object) in equality comparisons. The m-out-of-n threshold voting problem is then converted to m-out-of-$2^{\lfloor \log n \rfloor+1}$ voting on the padded list. 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. It is tempting to make the additional $2^{\lfloor \log n \rfloor+1} - n$ objects 'wild' so as to match any other object. It might then be argued (as was done in an early draft of this paper) that this approach increases the vote tallies by the same amount for all objects, making it easy to offset the bias by correspondingly adjusting the threshold to $m' = m + 2^{\lfloor \log n \rfloor+1} - n$. The fallacy of this argument lies in the fact that the bounded size of the candidate set can no longer be guaranteed.

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 emulates n/q processors in the original n-processor version. 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 n/q slowdown. 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 2 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 [27]. 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 place an upper bound $v_{\max}$ on the input votes (weights). Then, the number of candidates having at least $t/2$ votes in each subset of $n/2$ elements is no more than $\lfloor (n v_{\max}/2)/(t/2) \rfloor \le p v_{\max}$. Thus, the preceding asymptotic analyses still apply if $v_{\max}$ is a constant. In particular, if the input weights are in $\{0, 1\}$, the time complexity of weighted threshold voting is the same as that of m-out-of-n voting. The performance penalty of this approach is illustrated by Example 1 where the size of the candidate set grows from $\lfloor 15/8 \rfloor = 1$ to $\lfloor 6 \times 4/8 \rfloor = 3$.

When the input object space is totally ordered, weighted threshold voting can be performed with the same efficiency as m-out-of-n voting, discussed in Example 9 and the paragraph preceding it. In Example 9, it would have been sufficient to initialize the elements of the partition vector to the input weights (swapping the elements of this vector whenever input pairs are swapped to produce the sorted order) and executing the rest of the algorithm unmodified. For example, the initial votes 1, 2, 1, 2, 2, 3, 1, 3 would be transformed to 1, 2, 1, 2, 3, 3, 1, 2 after the sort step, yielding the tallies –, –, 4, –, 5, 3, –, 3 following the partitioned parallel prefix sum computation.

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 reasonably efficient, though asymptotically suboptimal by a $\log n$ factor, on the PRAM, binary hypercube, certain fixed-degree hypercubic networks, and a wide array of hierarchical architectures. The algorithm is also suboptimal on k-D mesh architectures by a factor of k, but since k tends to be relatively small in practice (typically 2 or 3), this is less of a problem. As far as we know, this work represents the first discussion of parallel voting algorithms beyond simple majority voting.

In the special case of majority voting, it was shown that the algorithm can be adapted 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.

We showed that our algorithm can be adapted for efficient parallel weighted threshold voting in cases where the maximum weight is not too large. 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.

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 $O(\log^2 n)$ and $O(p \log n + \log^2 n)$ complexities obtained under different assumptions for the PRAM, hypercube, and related hypercubic networks, or the $O(k^2 n^{1/k})$ and $O(kp + k^2 n^{1/k})$ complexities for k-D mesh architectures, are in fact the best possible (lower bounds). Extension of the algorithm to more efficient weighted voting schemes, particularly when $v_{\max}$ 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 [17], efficient parallel algorithms for these cases remain to be developed.

REFERENCES

[1] Parhami, B. (1995) Multi-sensor data fusion and reliable multi-channel computation: unifying concepts and techniques. *Proc. Asilomar Conf. Signals, Systems and Computers*, Pacific Grove, CA, October, pp. 745–749.

[2] Parhami, B. (1996) Design of Reliable Software via General Combination of N-Version Programming and Acceptance Testing. *Proc. 7th Intl Symp. Software Reliability Engineering*, White Plains, NY, October, pp. 104–109.

[3] Iyengar, S., Sitharama, S., Kashyap, R. and Madan, R. (guest eds) (1991) Special section on distributed sensor networks. *IEEE Trans. Systems Man Cybernetics*, **21**, 1027–1031.

[4] Parhami, B. (1996) A taxonomy of voting schemes for data fusion and dependable computation. *Reliability Engng Syst. Safety*, **52**, 139–151.

[5] Brownrigg, D. R. K. (1984) The weighted median filter. *Commun. ACM*, **27**, 807–818.

[6] Freund, Y. (1995) Boosting a weak learning algorithm by majority. *Info. Computat.*, **121**, 256–285.

[7] Battiti, R. and Colla, A. M. (1994) Democracy in neural nets: voting schemes for classification. *Neural Networks*, **17**, 691–707.

[8] Dolev, D., Lamport, L., Pease, M. and Shostak, R. (1987) The Byzantine generals. In Bhargava, B. K. (ed.), *Concurrency Control and Reliability in Distributed Systems*. pp 348–369, Van Nostrand Reinhold, New York.

[9] Lorczak, P. R., Caglayan, A. K. and Eckhardt, D. E. (1989) A theoretical investigation of generalized voters for redundant systems. *Proc. Intl Symp. Fault-Tolerant Computing*, Chicago, IL, June, pp. 444–451.

[10] Su, S. Y. H., Cutler, M. and Wang, M. (1991) Self-diagnosis of failures in VLSI tree array processors. *IEEE Trans. Comput.*, **40**, 1252–1257.

[11] Parhami, B. (1994) Threshold voting is fundamentally simpler than plurality voting. *Intl J. Reliability, Quality Safety Engng*, **1**, 95–102.

[12] Boyer, R. S. and Moore, J. S. (1991) MJRTY—A fast majority vote algorithm. In Boyer, R. (ed.), *Automated Reasoning: Essays in Honor of Woody Bledsoe*. Kluwer Academic, Dordrecht, The Netherlands.

[13] Gries, D. (1990) A hands-in-the-pocket presentation of a k-majority vote algorithm. In Dijkstra, E. W. (ed.), *Formal Development of Programs and Proofs*. pp. 43–45, Addison-Wesley, Reading, MA.

[14] Misra, J. and Gries, D. (1982) Finding repeated elements. *Sci. Comp. Programm.*, **2**, 143–152,. See also a related correspondence item in *Comput. J.* (1992) **35**, 298.

[15] Campbell, D. and McNeill, T. (1991) Finding a majority when sorting is not available. *Comput. J.*, **34**, 186.

[16] Lei, C. -L. and Liaw, H. -T. (1993) Efficient parallel algorithms for finding the majority element. *J. Info. Sci. Engng*, **9**, 319–334.

[17] Parhami, B. (1991) The parallel complexity of weighted voting. *Proc. Intl Symp. Parallel and Distributed Computing and Systems*, Washington, DC, October, pp. 382–385.

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

[19] Parhami, B. (1994) Voting algorithms. *IEEE Trans. Reliability*, **43**, 617–629.

[20] Parhami, B. (1991) Voting networks. *IEEE Trans. Reliability*, **40**, 380–394.

[21] Leighton, F. T. (1994) *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, San Mateo, CA.

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

[23] Latifi, S., Azevedo, M. and Bagherzadeh, N. (1993) The star-connected cycles: a fixed-degree network for parallel processing. *Proc. Intl Conf. Parallel Processing*, **1**, 91–95.

[24] Ghose, K. and Desai, R. (1995) Hierarchical cubic networks. *IEEE Trans. Parallel Distrib. Syst.*, **6**, 427–435.

[25] Yeh, C. -H. and Parhami, B. (1996) Swapped networks: unifying the architectures and algorithms of a wide class of hierarchical parallel processors. *Proc. Intl Conf. Parallel Distrib. Systems*, Tokyo, June, pp. 230–237.

[26] Yeh, C. -H. and Parhami, B. (1996) Recursive hierarchical swapped networks: versatile interconnection architectures for highly parallel systems. *Proc. 8th IEEE Symp. Parallel Distrib. Processing*, New Orleans, LA, October, pp. 453–460.

[27] Cormen, T. H., Leiserson, C. E. and Rivest, R. L. (1990) *Introduction to Algorithms*. McGraw-Hill, New York.