THE IMPACT OF SYNCHRONOUS COMMUNICATION ON THE PROBLEM OF ELECTING A LEADER IN A RING

Greg N. Frederickson

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Nancy A. Lynch

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

(ABSTRACT)

We consider the problem of electing a leader in a synchronous ring of n processors. We obtain both positive and negative results.

On the one hand, we show that if processor ID's are chosen from some countable set, then there is an algorithm which uses only O(n) messages in the worst case.

On the other hand, we obtain two lower bound results: If the algorithm is restricted to use only comparisons of ID's, then we obtain an $\Omega(n \log n)$ lower bound for the number of messages required in the worst case. Alternatively, there is a (very fast-growing) function f with the following property. If the number of rounds is required to be bounded by some t in the worst case, and ID's are chosen from any set having at least f(n,t) elements, then any algorithm requires $\Omega(n \log n)$ messages in the worst case.

1. Introduction

Communication in a network can be performed in either a synchronous or asynchronous mode. How does the choice of communication mode affect the computational resources required to solve a problem? We examine this question by considering the problem of electing a leader in a ring-shaped network. In this problem there are n processors, which are identical except that each has its own unique identifier. At various points in time, one or more of the processors "wake up", and initiate their participation in an election to decide on a leader. The relevant resources for such a distributed computation are the total number of messages used and the amount of time expended from the time that the first processor wakes up.

The problem of electing a leader efficiently has been studied by a number of researchers [B,CR,DKR,GHS,HS,IR,L,P]. The best previous deterministic algorithms have used O(n log n) messages for either bidirectional rings [HS,GHS,B] or unidirectional rings [DKR,P]. These algorithms work for both the synchronous and asynchronous models, and use comparisons of ID's only. In addition, Burns has established a lower bound of $\Omega(n \log n)$ on the number of messages required if communication is asynchronous [B]. However, the proof in [B] does not extend to the case of synchronous communication. It is, therefore, quite natural to ask whether the $\Omega(n \log n)$ lower bound can be achieved in the synchronous case as well as the asynchronous, or whether there are algorithms that somehow make use of the synchrony to limit the number of messages transmitted.

We obtain both positive and negative answers to our question of whether synchrony helps. On the one hand, we show that if processor ID's are chosen from some countable set (such as the integers), then there is an algorithm which uses only O(n) messages in the worst case. The processors may initiate the algorithm at different rounds, and do not know the value of n. Our algorithm is thus an improvement on a probabilistic algorithm of [IR] that uses O(n) messages on average and assumes that the processors do know the value n. Unlike the earlier algorithms, our algorithm does not only

The work of the first author was supported by the National Science Foundation under grant MCS-8201083. The work of the second author was supported by the NSF under Grant No. MCS79-24370, and Advanced Research Projects Agency of the Department of Defense Contract #N00014-75-C-0661.

use comparisons on ID's - it uses the numerical value of the ID's to count rounds. However, the number of synchronous rounds used by our algorithm can be very large in the worst case. An algorithm similar to ours has been developed independently by Vitanyi and appears in [V].

On the other hand, we show that both the departure from the comparison model, and the possibility of using a large number of rounds, are necessary in order to obtain a linear communication algorithm. More specifically, if the algorithm is restricted to use only comparisons of ID's, then we obtain an $\Omega(n \log n)$ lower bound for the number of messages required in the worst case. Alternatively, if the number of rounds is required to be bounded by some t in the worst case, then there is a (very fast-growing) function f(n,t) which has the following very interesting property. If ID's are chosen from any set T having at least f(n,t) elements, then any t-bounded algorithm requires $\Omega(n \log n)$ messages in the worst case. (In particular, if t is a function of n, say t(n), then any t(n)-bounded algorithm for a set T with at least f(n,t(n)) elements exhibits the given lower bound on messages.) We achieve this result by giving a transformation from any algorithm in what we call free form, over such a set T, to a comparison-based algorithm. (The ideas for this transformation are derived from earlier work of Snir [S1].) Both of our lower bound results hold even in the case that the number of processors in the ring is known to each processor, and all the processors are known to start at the same round.

2. The Algorithm

In this section, we present an algorithm for electing a leader in a synchronous ring, that uses O(n) messages, but may require a very large number of rounds. The elected processor (and no other processor) eventually enters one of a set of distinguished "elected" states. The total number of messages ever used (including any messages which might be sent after the winner is elected) is O(n). The algorithm presented is for a unidirectional ring, with communication assumed to be counterclockwise. Of course, a variant will work on a bidirectional ring. We assume that the unique ID of each processor is an integer. This assumption is reasonable if communication is implemented by transmitting packets of bits. In the description of the algorithm, we shall refer to the processor with ID i as "processor i".

At the beginning of the algorithm, each processor which chooses to participate in the election (henceforth called a "participating processor") spawns a *message process*, which moves around the ring, carrying the ID of the originating processor. The message process is charged one message for each edge which it traverses.

Our algorithm uses several ideas. The first is that message processes that originate at different processors are transmitted at different rates: the message process carrying processor ID i travels at the rate of one message transmission every 2^i rounds. (More specifically, each processor delays for $2^i - 1$ rounds before transmitting message process i.) Any slower message process that is overtaken by a faster message process is killed. Also, a message process carrying ID i arriving at processor j is killed if $j < i$ and processor j has also spawned a message process. A message process which returns to its originator causes that originator to become elected.

Suppose that all participating processors were to wake up at the same round. The strategy above would then guarantee that the total number of messages is O(n). For, assume i is the smallest ID of any participating processor. Message process i traverses all edges, for a total cost of n. Consider any other message process, j. During message process i's circuit, either message process i overtakes message process j, or else message process j reaches processor i; in either case, message process j is killed by the time i's circuit is completed. Because of the different rates of travel, message process j could travel at most distance $n/(2^{j-i})$ during the time i travels distance n. Thus, the message process carrying the smallest ID, i, must use more messages than all others combined. Since message process i uses n messages, the total number of messages expended would be less than 2n.

However, this variable rate of transmission scheme is by itself not enough to realize O(n) messages, in the case that not all participating processors wake up at the same time. The processors with smaller ID's could wake up correspondingly later, and spawn message processes that would chase and ultimately overtake the slower message processes, but not before $\Omega(n)$ messages had been expended by each of $\Omega(n)$ message processes.

The second idea is to have a preliminary phase, before the variable rate phase begins. In this phase, all message processes travel at the same rate, one message transmission per round. When a processor decides it wants to participate, it spawns its message process and sends it off to its neighbor. The message process is transmitted around the ring, until it

encounters the next participating processor. At this point, the message process continues into the second phase, moving at its variable rate, and acting as previously described. If a processor awakens after a message process has already passed by, then that processor will not participate in the election.

We now show that the total number of messages used in all phases of the computation is less than 4n.

> **Lemma 1:** After no more than n rounds from the time the first processor awakens, the message process of the eventual winner enters its second phase.
>
> **Proof:** If i and j are two processors in the ring, let d(i,j) denote the counterclockwise distance from i to j. Let i be the first processor to awaken, and let j be the eventual winner. Then a message process must arrive at j by the end of round d(i,j). Therefore, message process j must traverse its first edge by round d(i,j) + 1. Then message process j enters its second phase by the time it reaches i (or some closer participating processor), at most d(j,i) rounds later, i.e. by the end of round n. ∎

We now divide the messages into three categories, and bound each category separately. The categories are (1) the first phase messages, (2) the second phase messages sent before the eventual winner enters its second phase, and (3) the second phase messages sent after the eventual winner enters its second phase. Let i denote the eventual winner.

For (1), it is easy to see that the total number of first phase messages is exactly n. Next, consider (2). Lemma 1 implies that at most n rounds need to be considered. Moreover, message process i sends no second phase messages during the rounds under consideration. The smallest possible ID for the processors which are not eventual winners is 1, and the maximum number of second phase messages for message process j in these rounds is $n/(2^j)$. Thus, the total number of messages sent for all the message processes in these rounds is less than n.

Finally, consider (3). The argument is similar to the one used for the case in which all processors awaken at the same round. That is, message process i makes a circuit, for a total cost of n. Consider any other message process j. During message process i's circuit, either message process i overtakes message process j, or else message process j reaches processor i; in either case, message process j is killed by the time i's circuit is completed. Because of the different rates of travel, message process j could send at most $n/(2^{j-1})$ phase two messages during the time i travels distance n. Thus, as before, the total number of messages expended is less than 2n. The total number of messages for all categories is less than 4n.

Although the number of messages is quite small, the time required may be rather large. It is easy to see that the number of rounds expended is $O(n\,2^i)$, where i is the ID of the eventual winner. Thus, we obtain:

> **Lemma 2:** There is an algorithm which elects a leader in a synchronous ring of n processors using fewer than 4n messages, and $O(n\,2^i)$ time, where i is the ID of the eventual winner.

The bound of 4n messages for the algorithm above is reasonably tight. Consider the following example, where f(n) = log n - log log n. Let processor 1 be one link from processor 0, and let processor k, k = 2,...,f(n), be

$$k + \lfloor (2^{k-1} - 2)n/(2^{k-1} - 1) \rfloor$$

links from processor 0. Let processors 1 and 2 awaken at round 1, and each processor k, k = 3,...,f(n), awaken the round before it would be visited by a first phase message. Similarly, let processor 0 awaken the round before it would be visited. Then processor k, k = 1,...,f(n), will start its second phase at round $2 + \lfloor (2^k - 2)n/(2^k - 1) \rfloor$, and will traverse at least $n/(2^k - 1) - 2$ links before any message process overtakes it. There will be n first phase messages, at least

$$\Sigma_{k=1,\ldots,f(n)} (n/(2^k - 1) - 2)$$

second phase messages for processors k = 1,...,f(n), and n - 1 second phase messages for processor 0. For large n, this gives slightly more than 3.6n messages in all.

It is possible to reduce the number of messages at the expense of the number of rounds by using powers of c, for any constant c > 1, rather than powers of 2. As before, there will be exactly n messages in category (1). In category (2), there will be fewer than $\Sigma_{j=1,\ldots,\infty} n/c^j = n/(c-1)$ messages, while in category (3), there will be fewer than $\Sigma_{j=0,\ldots,\infty} n/c^j = nc/(c-1)$ messages. Thus, we obtain an algorithm which elects a leader in a synchronous ring of n processors using fewer than $2cn/(c-1)$ messages, and using at most $O(n\,c^i)$ rounds, where i is the ID of the eventual winner. Moreover, the leader elected by the algorithm is guaranteed to be one of the participating processors.

If we are willing to allow any processor to become elected, rather than just the participating processors, then it is possible to retain the $2cn/(c-1)$ message bound, while reducing the time to $O(n\,c^i)$, where i is the minimum ID of all processors in the ring. The basic idea is to allow each processor to awaken and begin its algorithm (spawning its message process) as

soon as it receives any message from its neighbor, if it has not already awakened on its own. We thus obtain:

> **Theorem 3:** Let $c > 1$. There is an algorithm which elects a leader in a synchronous ring of n processors using fewer than $2cn/(c-1)$ messages, and $O(n\, c^i)$ time, where i is the smallest ID of any of the processors in the ring.

We can also elect a leader from among the set of participating processors, while maintaining the O(n) message behavior and the dependence of the time on the smallest ID in the ring. All we need to do is to follow an algorithm which elects an arbitrary processor with an additional phase which elects a participating processor. In this additional phase, the originally elected processor just originates a message which circulates twice around the ring, determining the participating processor with the smallest ID.

Note that the algorithm works correctly in the case where communication is purely asynchronous. It is only its complexity that depends on the synchrony. In the general, asynchronous, case, the algorithm is essentially the same as that of [CR], and so exhibits a worst-case message behavior which is $O(n^2)$.

3. Framework for Lower Bound Proofs

In this section, we describe the assumptions we use for our lower bound results. We require a formal model for the lower bound results; we present the necessary definitions in this section as well. Finally, we define a special kind of algorithm called a "free" algorithm, and show that there is no loss of generality in restricting to free algorithms.

3.1. Assumptions

We assume that the communication is bidirectional. (Our proof may easily be adapted for unidirectional rings.) We assume that the value of n is a power of 2, and is known by each processor. All processors are assumed to awaken at the same time, round 1.

For the algorithm, we counted the total number of messages sent during an execution, including those sent after a processor got elected. For our lower bound results, on the other hand, we measure only the messages sent up to the point where a processor becomes elected.

Except for the restriction to powers of 2, our assumptions serve to strengthen the model, and hence the lower bound results.

3.2. Ring Algorithms

Each processor is modelled as an automaton that behaves as follows. At each round, each processor examines its state and decides whether to send a message to each of its neighbors, and what message to send. Then each processor receives any messages sent to it in that round. Each processor uses its current state and these new messages to update its state.

We now introduce formal definitions. An *ID space* is any totally ordered set. In this paper, T will denote an ID space. Let A denote a finite *decision alphabet*. Let M denote an alphabet of possible messages.

A *ring algorithm* over T and A is an automaton (Q,I,D,μ,δ), where

- Q is a set of states,
- $I \subseteq Q$ is the set of *initial states*, partitioned into nonempty sets I_t, one for each $t \in T$,
- $D \subseteq Q$ is the set of *decision states*, partitioned into D_a, $a \in A$, the set of a-decision states,
- μ is a *message generation function*, mapping $Q \times \{left,right\} \rightarrow M \cup \{null\}$,

and

- δ is a *transition function*, mapping $(M \cup \{null\}) \times Q \times (M \cup \{null\}) \rightarrow Q$.

The decision states are the means by which the automata produce output. We assume that the various sets D_a of decision states are "closed" under the operation of the transition function. Thus, once a decision has been reached, the same decision must persist.

The mapping μ decides, for each of the automaton's two neighbors, whether or not a message is to be sent, and in the former case, which message is to be sent. The mapping δ determines a new state from the old state and any messages arriving from the automaton's two neighbors.

3.3. Executions

We number the processors in the ring counterclockwise, as 0,...,n-1. We count indices modulo n. For the remainder of this paper, "processor i" will denote the processor numbered i in this counterclockwise numbering. We let $\langle n \rangle$ denote the set of integers {0,....,n-1}.

A *configuration* of width n is an n-tuple of elements of Q, representing the states for the n processors 0,...,n-1, in order. A *message vector* of width n is an n-tuple of ordered pairs of elements of M U {null}. It represents the messages sent left and right by each of the n processors.

An *execution* of width n is an infinite sequence of triples (C_1,N,C_2), where C_1 and C_2 are configurations and N is a message vector, all of width n. An *execution fragment* is any finite prefix of an execution. We require executions to satisfy several properties.

First, the initial configuration must consist of initial states. Each execution and execution fragment therefore has an *ID vector* in T_n which is the vector of T-values represented by the vector of initial states in the initial configuration. That is, if component i of the initial configuration is in I_t, then component i of the ID vector is t. We require that the different components of the ID vector of an execution all be distinct. (This condition models the distinctness of the processors' ID's.) We also require that each triple in an execution be "consistent" with the message generation and transition functions. Finally, the configurations in consecutive triples must "match up".

Each execution has a *decision vector* in $(A \cup \{null\})^n$, representing the eventual decisions made by all the component processors. That is, if component i of the configurations in the execution is eventually in D_a, then component i of the decision vector is a. If component i never enters any D_a, then component i of the decision vector is null.

A *message instance* is a quadruple (r,i,m,d), where r is a nonnegative integer denoting a round number, i is a processor index, $m \in M \cup \{null\}$, and $d \in \{left,right\}$. A message instance (r,i,m,d) is said to *occur* in execution (or execution fragment) e provided that in e, at round r, processor i sends message m in direction d.

The following definitions allow us to describe information flow via nonnull messages. For nonnegative integer k, we define a *right k-chain* in execution (or execution fragment) e to be a sequence s =

$$(r_1,i,m_1,right),(r_2,i+1,m_2,right),\ldots,(r_k,i+k-1,m_k,right)$$

of message instances occurring in e, where the rounds r_j are strictly increasing, and the m_j are nonnull messages. In this case, we say that the k-chain s *leads* to processor i + k. Symmetric definitions are made for left k-chains.

Now, we define our complexity measures. For any execution e, let *finishtime(e)* denote the number of the first round after which the eventual decisions in e have all been made (i.e. each component which has a nonnull entry, a, in the decision vector, is in a state in D_a). Let *messages(e)* denote the number of messages sent during e, up to and including round finishtime(e). We say that an algorithm requires no more than time t provided that finishtime(e) $\leq$ t for all executions, e. We say that the algorithm uses no more than s messages provided that messages(e) $\leq$ s for each execution, e.

3.4. Problems

Now, we consider the sense in which a ring algorithm solves a problem. A *problem* of width n over T and A is a mapping from length n vectors of distinct values in T to subsets of $(A \cup \{null\})^n$. A problem represents, for each particular ID vector, the allowable decision vectors.

For the problem of electing an arbitrary leader, we define the mapping so that it assigns to any vector, the set consisting of all vectors with exactly one 1, and all other positions null. For the problem of electing the processor with the minimum ID, the mapping would assign to any vector, the set consisting of the single vector which has 1 in the position corresponding to the minimum value in T and null elsewhere.

Two length n vectors, x and y, of T-elements are said to be *order-equivalent* provided that the elements in corresponding positions in x and y satisfy the same ordering relations. That is, for each pair of positions, i and j, we have $x_i < x_j$ exactly if $y_i < y_j$, and similarly for the relations = and >. A problem, P, is *order-invariant* provided that whenever two vectors, x and y, over T are order-equivalent, their images, P(x) and P(y) are identical, i.e. exactly the same set of vectors is permitted as output. The problem of electing a leader and the problem of electing the processor with the minimum ID are both order-invariant.

We say that a ring algorithm over T and A *solves* a problem, P, of width n, provided that $|T| \geq n$, and that for each execution, e, of the algorithm, the following holds. If e's ID vector is x then e's decision vector is an element of the set P(x) of allowable output vectors.

3.5. Free Algorithms

It will be convenient to assume that algorithms are in a particular "free" form, in which the states of processors record the initial ID and the history of messages received, and all messages contain the entire state of the sending processor. We show in this subsection that we can assume such a form.

The most natural way to represent such history information is by means of LISP S-expressions. The S-expressions that arise during computation are of a special type. The atoms are $t \in T$ and NIL. The *well-formed* S-expressions are just the following: (1) the elements of T, and (2) those of the form (s_1,s_2,s_3), where s_2 is a well-formed S-expression, and s_1 and s_3 are either well-formed S-expressions or NIL.

An algorithm is *free* provided that its state set and its message alphabet are both just the set of well-formed S-expressions. Also, the initial states are just the atoms in T (the ID's). Moreover, the message sent in either direction from any state q is either just the state q or null, and the new state arising from state q with messages m_1 and m_2 arriving from the left and right respectively is just the S-expression (m_1,q,m_2). (If either m_1 or m_2 is null, then we use NIL.)

The parts of the algorithm which are still undetermined are whether, for each state and each direction, a message is sent, which of the states are decision states, and how the decision states are partitioned into D_a for various a. For a free algorithm, it is helpful to define a subsidiary message function, μ', which maps $Q \times \{left,right\} \rightarrow \{yes,no\}$, depending on whether a message is supposed to be sent in the corresponding direction from that state. (If a message is sent, its actual contents are determined by the state of the sender.)

The following lemma says that for the complexity measures considered in this paper, there is no loss of generality in restricting attention to free algorithms.

> **Lemma 4:** Let $\mathcal{A}$ be a ring algorithm, over T and A, which solves problem P using no more than time t, and no more than s messages. Then there is a free algorithm, $\mathcal{A}'$, over T and A, which solves P using no more than time t, and no more than s messages.
>
> **Proof:** We use notation Q, etc. from the definition of a ring algorithm, to refer to algorithm $\mathcal{A}$. For each $t \in T$, let q(t) be a designated initial state in I_t.
>
> We define eval(s), for each well-formed S-expression, s, to be a particular state in Q. We do this inductively. First, define eval(t), for t an atom, to be q(t). Next, if $s = (s_1,s_2,s_3)$, then define eval(s) to be $\delta(a_1,eval(s_2),a_3)$, where a_1 = null if s_1 = NIL, and $a_1 = \mu(eval(s_1),right)$ otherwise, while a_3 = null if s_3 = NIL, and $a_3 = \mu(eval(s_3),left)$ otherwise.
>
> Now, we allow $\mathcal{A}'$ to send a message left from state s provided that $\mathcal{A}$ sends a message left from state eval(s), and analogously for messages sent right. Similarly, state s is an a-decision state exactly if state eval(s) is an a-decision state.
>
> It should be clear that $\mathcal{A}'$ "simulates" the behavior of $\mathcal{A}$. Therefore, it is straightforward to check that $\mathcal{A}'$ also solves P in time t, and uses no more than s messages.
>
> ∎

If we were interested in counting, say, the total number of bits of communication, it would not be sufficient to restrict attention to free algorithms, since the algorithm transformation described in the preceding lemma can cause a large increase in the size of messages.

We require one more simple but important lemma about free algorithms. This lemma imposes a limit on the propagation of information by failing to send messages.

> **Lemma 5:** Let $\mathcal{A}$ be any free algorithm which has no right k_1-chains and no left k_2-chains leading to processor i in execution fragment e. If a is a T-value which is in the state of processor i at the end of e, then a was the initial value of some processor j, where $i - k_1 + 1 \leq j \leq i + k_2 - 1$.
>
> **Proof:** Straightforward. ∎

4. Lower Bound for Comparison Algorithms

In this section, we restrict attention to algorithms which use comparisons only. We present our first lower bound, of $\Omega(n \log n)$, for the number of messages required for a comparison algorithm, to elect a leader in a synchronous ring.

4.1. Comparison Algorithms

In this subsection, we define "comparison algorithms".

We say that two S-expressions are *order-equivalent* provided that they are identical except for the particular atoms which occupy various positions within the expressions, and corresponding atoms satisfy the same ordering relations in the two expressions. A free algorithm is a *comparison algorithm* provided that if s and s' are order-equivalent well-formed S-expressions, then processors with states q and q' transmit messages in the same direction or directions and are in the same set of decision states (if any). (That is, $\mu'(q,left) = \mu'(q',left)$, $\mu'(q,right) = \mu'(q',right)$, and for each $a \in A$, q is in D_a exactly if q' is in D_a. Recall that μ' is the subsidiary

message function which decides whether or not to send a message, but does not say exactly which message is sent.)

4.2. Preliminary Results

In this subsection, we assume that n is a power of 2, and let T^* be the ID space consisting of the set $\langle n \rangle$, with the usual ordering.

For any $m \leq \log n$, define a function *m-high* from $\langle n \rangle$ to $\langle 2^m \rangle$ so that m-high(i) is the integer represented by the m high-order bits of i. Extend the m-high mapping to the set of S-expressions over T^*, by replacing every non-NIL atom, i, with m-high(i).

For $i \in \langle n \rangle$, let *reverse(i)* denote the integer whose binary representation is the reverse of the binary representation of i. We assign processor ID's so that the values are arranged consecutively, counterclockwise around the ring in order of increasing reverse(i) values. Thus, for each $m < \log n$, the values repeatedly cycle through the 2^m possible patterns of m high-order bits. This pattern exhibits a large amount of "local symmetry" which we exploit for our results.

For the remainder of this subsection, assume that $\mathcal{A}$ is any particular comparison algorithm over T^*. Also assume that e is an execution fragment of some execution of $\mathcal{A}$ whose ID vector is given by the pattern described above.

The next lemma says that, if the sum of the lengths of the maximum right chain and maximum left chain leading to a processor is strictly less than 2^m, then all ordering information about the T^*-elements which the processor has as atoms in its state is determined solely by the m high-order bits.

> **Lemma 6:** Let $m < \log n$. Assume that, in execution fragment e, the sum of the lengths of the maximum right chain and the maximum left chain leading to a processor, i, is less than 2^m. Let a and b be any two T^*-elements occurring in processor i's state at the end of e. Then
>
> (a) m-high(a) = m-high(b) if and only if a = b,
>
> (b) m-high(a) < m-high(b) if and only if a < b,
>
> and
>
> (c) m-high(a) > m-high(b) if and only if a > b.
>
> **Proof:** Lemma 5 and the distribution of ID's shows that m-high(a) = m-high(b) implies a = b. The other cases follow immediately. ∎

As simple consequence of the preceding lemma is the following.

> **Lemma 7:** Let $m < \log n$. Assume that, in execution fragment e, the sum of the lengths of the maximum right chain and the maximum left chain leading to processor i is less than 2^m, and similarly for processor j. Let q and q' be the states of processors i and j, respectively, after e. Assume that m-high(q) = m-high(q'). Then q and q' are order-equivalent.
>
> **Proof:** Follows easily from Lemma 6. ∎

The following key claim shows how limited message propagation forces certain corresponding processors to be in corresponding states.

> **Lemma 8:** Let $m < \log n$. Assume that, in execution fragment e, for each processor, the sum of the lengths of the maximum right chain and the maximum left chain leading to the processor is less than 2^m. Let q and q' be states of processors i and $i + 2^m$, respectively, after e has been executed.
>
> Then (a) m-high(q) = m-high(q').
>
> (b) If $a \in A$, then q is in D_a exactly if q' is in D_a.
>
> (c) $\mu'(q,\text{left}) = \mu'(q',\text{left})$ and $\mu'(q,\text{right}) = \mu'(q',\text{right})$.
>
> **Proof:** We proceed by induction on the length of e.
>
> Base: e is of length 0
>
> Initial states consist only of T^* elements, and the chosen pattern ensures that processors which are exactly distance 2^m apart have the same m high-order bits. This shows (a). Then (b) and (c) follow because $\mathcal{A}$ is a comparison algorithm.
>
> Inductive Step: e is of length > 0
>
> By inductive hypothesis, processors i and $i + 2^m$ are in m-high equivalent states prior to the last step of e, as are processors $i - 1$ and $i - 1 + 2^m$, and processors $i + 1$ and $i + 1 + 2^m$. Moreover, at the last step of e, processors $i - 1$ and $i - 1 + 2^m$ either both generate right messages or else neither does. Similarly, at the last step of e, processors $i + 1$ and $i + 1 + 2^m$ either both generate left messages or else neither does. Therefore, q and q' are easily seen to be m-high equivalent, showing (a). Then Lemma 7 implies that q and q' are order-equivalent. Then (b) and (c) follow because $\mathcal{A}$ is a comparison algorithm.
>
> ∎

4.3. The Main Result

In this section, we prove the main lower bound theorem. We require one more (fairly obvious) lemma about comparison algorithms.

Lemma 9: Let T and T' be arbitrary ID spaces, n any integer. Assume that $\mathcal{A}$ is a comparison algorithm over T which elects a leader in a ring of size n and uses at most s messages. Then there exists a comparison algorithm $\mathcal{A}'$ over T' which elects a leader in a ring of size n and uses at most s messages.

Proof: We define $\mathcal{A}'$ as follows. For each well-formed S-expression, L', with atoms in T', let L be an order-equivalent S-expression with atoms in T. Define the values of the message decision function and the decision status for L' to be the same as the corresponding values for L. The fact that $\mathcal{A}$ is a comparison algorithm assures that this definition is unique.

Any input vector, y, of ID's in T' is order-equivalent to some input vector, x, of ID's in T. The computation of $\mathcal{A}'$ on input y therefore imitates the computation of $\mathcal{A}$ on input x, sending messages at the same times, and entering decision states at corresponding times. Since a leader is elected in the computation of $\mathcal{A}$ on x, it follows that a leader is elected in the computation of $\mathcal{A}'$ on y, and the message requirements are bounded by the corresponding requirements for $\mathcal{A}$ on x.

∎

Now, we prove the main result.

Theorem 10: Assume n is a power of 2. Let $\mathcal{A}$ be a comparison algorithm over an arbitrary ID space, T, which elects a leader in a synchronous ring of size n. Then there is an execution, e, of $\mathcal{A}$ for which messages(e) $\geq$ (n/2)(log n + 1).

Proof: Lemma 9 implies that it suffices to consider T = T^*. Let e be the execution on the distribution of ID's given in the preceding subsection. Let e' be the execution fragment of e which terminates just when the elected processor enters a state in D_1 (i.e. an elected state).

We first claim that in e', some processor i must have the sum of the lengths of the maximum right-chain and the maximum left-chain leading to it, at least n/2. For if not, then Lemma 8 implies that any pair of diametrically opposed processes would have the same decision status at the end of e', making it impossible to elect a leader.

For any prefix, e'', of e', let *maxright(e'')* denote the maximum length of any right chain in e'', and analogously for *maxleft(e'')*. Let *sum(e'')* denote maxright(e'') + maxleft(e''). The claim above implies that sum(e') $\geq$ n/2. Thus, sum(e'') starts out with a value of 0, when e'' is the empty sequence, and increases until it reaches at least n/2, when e'' = e'.

Consider any step at which sum(e'') increases. It is only possible for maxright(e'') to increase by 1 at one step, and similarly for maxleft(e''). Assume, for some particular m < log n, that sum(e'') < 2^m at the beginning of this step. We consider three cases.

Case 1: maxright(e'') increases by 1 and maxleft(e'') does not increase.

Then someone sends a message to the right at this step. Therefore, by Lemma 8, all processors which are separated from this processor by multiples of 2^m also send messages to the right at this step. Thus, at least n/(2^m) messages are sent to the right at this step.

Case 2: maxleft(e'') increases by 1 and maxright(e'') does not increase.

An argument similar to the one for Case 1. shows that at least n/(2^m) messages are sent to the left at this step.

Case 3: Both maxright(e'') and maxleft(e'') increase by 1 at this step.

Then a similar argument to the previous two cases shows that at least n/(2^m) messages are sent right, and also at least n/(2^m) messages are sent left at this step.

Thus, a cost of at least n/(2^m) messages is incurred for each increase of 1 in sum(e''), whenever the sum before the increase is less than 2^m. Therefore, increasing sum(e'') from 0 to 1 requires n messages to be sent. Increasing sum(e'') from 1 to 2 requires n/2 additional messages, from 2 to 3 requires at least n/4 additional messages, from 3 to 4 requires at least n/4 additional messages, from 4 to 5 requires n/8 messages, etc. In other words, n/2 messages are required to increase sum(e'') from 2 to 4, from 4 to 8, and in general, from any 2^m to 2^{m+1}. So the total number of messages required to increase sum(e'') from 0 to n/2 is at least n + (n/2)(log n - 1) = (n/2)(log n + 1), as required.

∎

5. Lower Bound for Time-Bounded Algorithms

In this section, we prove our lower bound for time-bounded algorithms. We use the lower bound for comparison algorithms to do this. First, we show how to map from time-bounded algorithms to comparison algorithms. This result, presented in the paracomputer model, is due to Snir [S1]. (Snir [S2] credits Yao [Y] with inspiration for this result.) For completeness, we present a careful proof in our setting, even though basically the same proof appears in [S1]. We then infer the lower bound for time-bounded algorithms.

5.1. Definitions

In order to map from time-bounded to comparison algorithms, we require definitions describing the behavior of an algorithm within a bounded amount of time.

We say that a free algorithm is a *t-comparison algorithm* provided that both of the following conditions hold.

(1) If s and s' are order-equivalent S-expressions of parenthesis depth at most t-1, then $\mu'(s,left) = \mu'(s',left)$ and $\mu'(s,right) = \mu'(s',right)$.

(2) If s and s' are order-equivalent S-expressions of depth at most t, and $a \in A$, then s is in D_a exactly if s' is in D_a.

During execution of a free algorithm, the S-expressions which appear as states at the end of any round t have depth exactly t. Thus, this definition says that the algorithm behaves as a comparison algorithm up to the end of the first t rounds.

We also add the qualifier "on inputs from U" to this definition, provided that the appropriate conditions hold for those S-expressions which use atoms chosen from the set U.

5.2. Mapping a Time-Bounded Algorithm to a Comparison Algorithm

In this subsection, we show how to convert a time-bounded algorithm to a comparison algorithm. The first step is to show that any free algorithm behaves as a comparison algorithm on a subset of its inputs. For the first lemma, we use a particular fast-growing function f(n,t). The precise definition of f depends on Ramsey's Theorem, and is implicit in the proof of the lemma.

Lemma 11: Fix n, t. Let $\mathcal{A}$ be any free algorithm over ID space T and alphabet A, where T has at least f(n,t) elements. Then there exists a subset U of T, of size at least n, such that $\mathcal{A}$ is a t-comparison algorithm, on inputs from U.

Proof: For any set V, let $\mathcal{L}(V)$ denote the set of well-formed S-expressions over atoms in V, of depth up to t. The order-equivalence relation splits the set $\mathcal{L}(T)$ into finitely many equivalence classes, $E_1,\ldots,E_k$. We build a sequence of sets U_j, each contained in the previous, such that the following property is satisfied. If s and s' are two S-expressions in $\mathcal{L}(U_j) \cap E_j$, then:

(1) $\mu'(s,left) = \mu'(s',left)$ and $\mu'(s,right) = \mu'(s',right)$,

and

(2) for any a in A, $s \in D_a$ exactly if $s' \in D_a$.

Letting $U = U_k$ then yields the needed result.

Initially, let $U_0 = T$. We now describe how to generate U_j, assuming that U_{j-1} has been defined.

There are only $c = 4(|A| + 1)$ possible combinations of choices that can be made for each expression in E_j: whether a message is sent left, whether a message is sent right, and the decision status.

Suppose that the expressions in E_j contain m distinct atoms. For each size m subset, X, of U_{j-1}, there is a unique expression, L(X), in E_j containing the elements of X as atoms. We "color" X with a color corresponding to the choices made for the expression L(X). Thus, we obtain a c-coloring of the collection of size m subsets of U_{j-1}.

According to Ramsey's Theorem [BE], there is a subset of U_{j-1}, which we call U_j, such that all m-element subsets of U_j are colored the same color; U_j can be chosen to be of any predetermined size provided that U_{j-1} is sufficiently large.

This U_j is easily seen to have the needed properties.

∎

The next lemma gives the mapping from free time-bounded algorithms to comparison algorithms.

Lemma 12: Fix n and t. Let $\mathcal{A}$ be a free algorithm over ID space T and alphabet A, where T has at least f(n,t) elements. Let P be an order-invariant problem, of size n. If $\mathcal{A}$ solves P in t rounds, using at most s messages in the worst case, then there exists a comparison algorithm $\mathcal{A}'$, which solves P in t rounds, using at most s messages in the worst case.

Proof: The proof is similar to that of Lemma 9. We are going to construct $\mathcal{A}'$ which "simulates" the behavior of $\mathcal{A}$ for the first t rounds. Since $\mathcal{A}$ arrives at all the proper decisions by the end of round t, $\mathcal{A}'$ will also do so. Thus, we can allow $\mathcal{A}'$ to carry out only trivial activity after round t.

All the states which arise in algorithm $\mathcal{A}$ up to the end of round t have expression depth which is at most t. Thus, we define the message generation function to yield "null" for any expression of depth greater than or equal to t. In order to make sure that the algorithm satisfies the required "closure" condition for sets of decision states, we define a well-formed S-expression of depth greater than t to be in D_a provided that its middle component is in D_a. It is clear that these conventions are consistent with the fact that $\mathcal{A}'$ is a comparison algorithm.

Now we must describe the message decision function of $\mathcal{A}'$ for expressions of depth up to and including t - 1, and decision status for expressions of depth up to and including t.

Lemma 11 says that there exists a subset, U, of T, of size at least n, such that for inputs from U, the algorithm $\mathcal{A}$ is a t-comparison algorithm. Consider any S-expression, L, of depth less than t, with atoms in T. Define the value of the message decision function on this expression to be that of the message decision function of $\mathcal{A}$ on any S-expression, L', with atoms from U, which is order-equivalent to L. The fact that $\mathcal{A}$ is a t-comparison algorithm on inputs from U ensures that this value is uniquely defined. Similarly, for any S-expression of depth at most t, with atoms in T, define membership in any D_a according to membership in D_a of any order-equivalent S-expression with atoms in U. It is obvious that $\mathcal{A}'$ is a comparison algorithm.

Now, we argue that $\mathcal{A}'$ solves P in t rounds. Any length n ID vector, y, over T, is order-equivalent to some ID vector, x, over U. The computation of $\mathcal{A}'$ on input y therefore imitates the computation of $\mathcal{A}$ on input x, up to the end of round t, sending messages at the same times, and entering decision states at corresponding times. Since $\mathcal{A}$ solves P in t rounds, the vector of states after t rounds of the computation of $\mathcal{A}$ on input x has the decision status of all components corresponding to some vector in $(A \cup \{null\})^n$ in the set allowed by P for input x. The vector of states after t rounds of the computation of $\mathcal{A}'$ on input y therefore has the decision status of all components corresponding to the same vector. Since P is an order-invariant problem, this vector is in the set allowed by P for input y. Therefore, $\mathcal{A}'$ solves P in t rounds.

Finally, we consider the number of messages sent before reaching final decision status. Say that $\mathcal{A}'$ on input y reaches its final decision status after round t'. It must be that $t' \leq t$. Then the computation of $\mathcal{A}$ on input x reaches its final decision status after round t' also. So the numbers of messages correspond as required. ∎

We can combine the immediately preceding result with Lemma 4 to obtain the following.

Lemma 13: Fix n and t. Let $\mathcal{A}$ be any algorithm over ID space T and alphabet A, where T has at least f(n,t) elements. Let P be an order-invariant problem, of size n. If $\mathcal{A}$ solves P in t rounds, using at most s messages in the worst case, then there exists a comparison algorithm $\mathcal{A}'$, which solves P in t rounds, using at most s messages in the worst case.

The immediately preceding result appears to be of much wider applicability than just to this work and Snir's. This result, or variants, should be very useful for the study of other order-invariant problems on many different kinds of computation models.

5.3. The Main Result

Finally, we present our lower bound for time-bounded algorithms.

Theorem 14: Fix n, t, where n is a power of 2. Let T be an arbitrary ID space with at least f(n,t) elements. Let $\mathcal{A}$ be any algorithm over T which elects a leader in a synchronous ring of size n, using no more than time t. Then there is an execution, e, of $\mathcal{A}$ for which messages(e) $\geq$ (n/2)(log n + 1).

Proof: Assume the contrary - that there exists an algorithm $\mathcal{A}$ over T which elects a leader in a synchronous ring of size n, using no more than time t, and using fewer than (n/2)(log n + 1) messages in the worst case. Then Lemma 13 implies that there exists a comparison algorithm which elects a leader in t rounds and uses fewer than (n/2)(log n + 1) messages in the worst case. However, this contradicts Theorem 10. ∎

6. Remaining Questions

There are several directions for further work.

First, the given bound is still not tight. The best known upper bound appears in [DKR,R], and is approximately 1.4 n log n. Our lower bound is approximately 1/2 n log n. It would be interesting to close this gap.

Second, the lower bounds in this paper rely on n being a power of 2. Unlike most other cases where such an assumption is made, in this case the assumption seems to be crucial. Whereas Burns' lower bound of Ω for the asynchronous case applies for all values of n, we do not know what happens in the synchronous case for non-powers of 2. It seems likely that the lower bound proof should extend in some way, but the extension appears to be nontrivial.

Third, it would be interesting to see whether the new techniques in this paper provide lower bounds for other order-invariant problems besides just election of a leader. Some preliminary work in this direction has already been carried out [GLTWZ].

Fourth, it would be interesting to consider results for election of a leader and other order-invariant problems in more general classes of graphs. For example, Angluin [A] characterizes graphs in terms of the possibility and impossibility of electing a leader, in the absence of unique identifiers. Our techniques might be useful for proving lower bounds for the number of messages required to elect a leader in various kinds of graphs, even if the processors do have unique ID's.

Acknowledgements:

The authors thank Cynthia Dwork for making us aware of the very interesting results of Snir, and noting their connection to our work. Thanks also go to Mike Fischer for several suggestions on improving the presentation.

References:

[A] D. Angluin, Local and Global Properties in Networks of Processors, *Proceedings of the 12th Annual ACM Symposium on Theory of Computing* (1980), pp. 82-93.)

[B] J. E. Burns, A formal model for message passing systems, TR-91, Indiana University (September 1980).

[BE] C. Berge, Graphs and hypergraphs, North-Holland, Amsterdam, 1973.

[CR] E. Chang and R. Roberts, An improved algorithm for decentralized extrema-finding in circular configurations of processes, (1979) 281-283.

[DKR] D. Dolev, M. Klawe and M. Rodeh, An O *(n* log *n)* unidirectional distributed algorithm for extrema finding in a circle, *J. Algorithms 3,3* (September 1982) 245-260.

[GHS] R. G. Gallager, P. A. Humblet and P. M. Spira, A distributed algorithm for minimum-weight spanning trees, *ACM Trans. Prog. Lang. Sys.* 5, 1 (January 1983) 66-77.

[GLTWZ] E. Gafni, M. Loui, P. Tiwari, D. West and S. Zaks, Lower bounds on common knowledge in distributed algorithms (ABSTRACT).

[HS] D. S. Hirschberg and J. B. Sinclair, Decentralized extrema-finding in circular configurations of processes, *Comm. ACM 23* (November 1980) 627-628.

[IR] A. Itai and M. Rodeh,

[L] G. LeLann, Distributed systems - toward a formal approach, *Information Processing 77*, North Holland, Amsterdam (1977) 155-160.

[P] G. L. Peterson, An O *(n* log *n)* unidirectional algorithm for the circular extrema problem, *Trans. Prog. Lang. Sys. 4*, 4 (1982) 758-762.

[S1] M. Snir, On parallel searching, Hebrew University of Jerusalem, Department of Computer Science, RR 83-21 (June 1983).

[S2] M. Snir, Personal communication (1983).

[V] P. Vitanyi, Distributed elections Archimedean Ring of Processors, This Proceedings.

[Y] A. Yao, Should tables be sorted? *J. ACM 28*, 3 (July 1981) 615-628.