Computing on an Anonymous Ring

HAGIT ATTIYA AND MARC SNIR

Hebrew University, Jerusalem, Israel

AND

MANFRED K. WARMUTH

University of California, Santa Cruz, California

Abstract. The computational capabilities of a system of n indistinguishable (anonymous) processors arranged on a ring in the synchronous and asynchronous models of distributed computation are analyzed. A precise characterization of the functions that can be computed in this setting is given. It is shown that any of these functions can be computed in $O(n^2)$ messages in the asynchronous model. This is also proved to be a lower bound for such elementary functions as AND, SUM, and Orientation. In the synchronous model any computable function can be computed in $O(n \log n)$ messages. A ring can be oriented and start synchronized within the same bounds.

The main contribution of this paper is a new technique for proving lower bounds in the synchronous model. With this technique tight lower bounds of $\Theta(n \log n)$ (for particular n) are proved for XOR, SUM, Orientation, and Start Synchronization. The technique is based on a string-producing mechanism from formal language theory, first introduced by Thue to study square-free words. Two methods for generalizing the synchronous lower bounds to arbitrary ring sizes are presented.

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*distributed networks*; C.2.5 [**Computer-Communication Networks**]: Local Networks—*rings*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*relations among models*; F.1.2 [**Computation by Abstract Devices**]: Models of Computation—*paralellism*; F.1.3 [**Computation by Abstract Devices**]: Complexity Classes—*relations among complexity measures*; F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting Systems—*parallel rewriting systems*

General Terms: Algorithms, Theory, Verification

Additional Key Words and Phrases: Communication complexity, coordination, distributed algorithms

1. *Introduction*

In this paper we study the complexity of computations on a distributed network of processors with a ring topology. In a distributed computation each processor starts with some initial state; it proceeds by sending messages to its neighbors, receiving

A short version of this work appeared in *Proceedings of the 4th Annual ACM Symposium on Principles of Distributed Computing* (Minacki, Canada, Aug. 5–7). ACM, New York, 1985, pp. 196–203.

The work of M. Warmuth was supported by the United States–Israel Binational Science Foundation, grant 2439/82, and the Office of Naval Research grant N00014-86-K0454.

Authors' present addresses: H. Attiya, Department of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel; Marc Snir, IBM T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598; M. Warmuth, Department of Computer Science, University of California, Santa Cruz, CA 95064.

messages from them, and updating its state accordingly, until it halts in a final state. The *complexity* of a computation is the number of messages sent by all processors in the worst case.

We are interested in problems in which the final state of each processor depends on the initial states of all the processors; when the computation halts, each processor has some "global information" on the initial configuration. Any such computation problem has at least linear complexity: each processor has to receive a message. If the activity of the processors is coordinated correctly, then any computable problem can be solved with a linear number of messages. Assume, for example, that the ring has a unique distinguished processor, the ring *leader*. The leader initiates a message; each processor appends to this message its own initial state and forwards the message; the leader receives back a message describing the entire ring; this message is forwarded around the ring. Each processor now has a complete description of the initial ring configuration and can compute the required answer locally.

A leader can be chosen whenever the initial configuration has a distinguished processor (the configuration is "centered," in the terminology of Angluin [1]). For example, if the processors have distinct labels, then the processor with the largest label can be elected as leader using $O(n \log n)$ messages [5, 8, 12]; $\Omega(n \log n)$ is also a lower bound for election [4, 7, 12].

In this paper we consider the situation in which no such distinguished processor exists—the processors are identical (*anonymous* ring). Some problems cannot be solved in this setting; for example, there is no way to break symmetry so as to choose a leader deterministically [1]. We investigate problems that can be solved deterministically on an anonymous ring, but where symmetry affects the cost of a solution.

The typical problems considered are those in which there are $\{0, 1\}$ initial states and processors reach a final state that is a Boolean function of the initial states, such as XOR or AND. Another problem, which is specific to the ring, is that of *orientation*. We assume that processors can tell their *left* from their *right*; however, their notions are not necessarily consistent. We want to orient the ring consistently, that is, have processors agree on what is left and what is right.

Since all problems can be solved with a linear number of messages with suitable coordination, information transfer arguments yield only trivial linear lower bounds; nonlinear lower bounds are obtained by taking into account the cost of achieving coordination. In a distributed computation on an anonymous ring the main problem is that of symmetry breaking: The state of a processor after few steps of the computation depends only on the initial configuration in a small neighborhood of that processor. If each neighborhood is replicated many times in the initial configuration, then whenever a processor sends a message, many other processors do so; "superfluous" traffic cannot be avoided. Tight lower bounds will be obtained by building "symmetric" initial configurations with many repetitions of local patterns; efficient algorithms are obtained by detecting symmetries as fast as possible.

All processors can acquire complete information on the initial ring configuration with a total of $O(n^2)$ messages: Each processor initiates a message containing its state, and these messages are forwarded n times. For the synchronous model, a classical election algorithm is modified to obtain an algorithm that enables each processor to acquire complete information on the initial ring configuration with a total of $O(n \log n)$ messages. A similar algorithm is used to orient a ring in $O(n \log n)$ messages. In the context of synchronous computations we also consider the problem of *start synchronization*: We assume that processors may "wake up" at arbitrary times; their clocks, however, are running at the same speed. The start synchronization algorithm presented enables all processors to reset their clocks simultaneously using $O(n \log n)$ messages. These results are presented in Section 4.

In Section 5 we present the first lower bounds: In the asynchronous model, at least $\Omega(n^2)$ messages are required to compute many elementary functions, in particular AND. As an interesting corollary we find that if input values are not unique, then extremum finding requires $\Omega(n^2)$ messages in anonymous rings (this result was proved independently by Itai [9]). Note that $O(n \log n)$ messages are sufficient when inputs are distinct. An $\Omega(n^2)$ lower bound on number of messages is also proved for orientation.

The rest of the paper is dedicated to the synchronous lower bounds. In Section 6 the general lower bound technique is presented. Using it we prove that $\Omega(n \log n)$ is a lower bound on the complexity of many elementary problems, such as computing XOR and orienting the ring. Section 7 is dedicated to extensions of the results to arbitrary ring lengths.

We conclude, in Section 8, with a discussion of the results.

2. *Definitions*

Consider a system of n processors, named $1, \ldots, n$, arranged on a bidirectional ring. Every processor i has communication channels with its two direct neighbors on the ring, *left*(i) and *right*(i). The processors need not be oriented consistently; it is possible either that *left*$(i) = i - 1$ and *right*$(i) = i + 1$, or that *left*$(i) = i + 1$ and *right*$(i) = i - 1$.[1]

An algorithm specifies the behavior of each processor, modeled as a state machine. The initial state of the processor is its input value. Two computation models are treated:

In the *synchronous* model all processors start the computation simultaneously and proceed in lock step (we later relax the assumption of simultaneous start). Message transfers and state transitions are clock driven: At each cycle a processor may send a message to its left neighbor and to its right neighbor, depending on its current state. Next the processor accepts messages sent by its neighbors, moves to a new state, and possibly halts, as a function of its previous state and the value of the communications from left and right. The computation terminates when all processors have halted; the state of a processor when it halts is its output.

In the *asynchronous* model the transmission of a message incurs an unpredictable but finite delay. Messages sent on a link are received in the order in which they are sent. State transitions are message driven: A processor receives one message at a time; whenever it receives a message, it possibly sends messages to its neighbors, moves to a new state, and possibly halts. A conceptual "start" message causes the first state transition at each processor.

We consider the case in which processors are indistinguishable; the indices are not available to the processors, and they all run the same algorithm. This is an *anonymous* ring. In a *labeled* ring, the indices are available to the processors, so that each may run a distinct algorithm.

The following notation is used:

The *orientation* of processor i is $D(i)$: $D(i) = 1$ if *right*$(i) = i + 1$, $D(i) = 0$ if *right*$(i) = i - 1$; $D = \langle D(1), \ldots, D(n) \rangle$ is the *ring orientation.* The ring is *clockwise oriented* if $D(i) = 1$ for all i, *counterclockwise oriented* if $D(i) = 0$ for all i. The

[1] Here and throughout the paper processor indices are taken modulo the ring size n; i and $n + i$ indicate the same processor.

ring is *oriented* if it is clockwise or counterclockwise oriented. A ring is oriented iff $i = left(right(i))$ for every processor i on the ring.

A processor does not "know" its orientation; $D(i)$ is not an input of the algorithm.

The initial input state of processor i is $I(i)$; $I = \langle I(1), \ldots, I(n)\rangle$ is the *ring input*; $R = \langle D(1), I(1), \ldots, D(n), I(n)\rangle$ is the *initial ring configuration.*

The output state of processor i when the computation halts is $O(i)$; $O = \langle O(1), \ldots, O(n)\rangle$ is the *ring output.*

A *problem* Π is a mapping that assigns to each initial configuration R a set $\Pi(R)$ of ring outputs, the set of *correct solutions* for R. An algorithm A *solves* the problem Π if a computation of A started on a configuration R ends with a ring output that is a correct solution for R, whenever such a solution exists.

Most problems we consider consist of computing a function f: A computation started with input I ends with each processor in state $f(I)$. For each initial configuration there is a unique correct solution. In the *orientation problem* each processor is required to orient its communication channels so that the whole ring becomes oriented. Formally, each processor computes a Boolean output, so that if the processors with output 1 switch their left and right connections, then the ring becomes oriented, either clockwise or counterclockwise. There are two correct solutions for each initial ring configuration: either $O(i) = D(i)$ for each i (counterclockwise orientation), or $O(i) = \overline{D(i)}$ for each i (clockwise orientation).

Our cost function will be either the number of messages sent or the number of bits sent, for some binary encoding of the messages. Lower bounds will be on the total number of messages sent, and algorithms will be analyzed by counting the total number of bits sent. All bounds are on worse case complexity.

In order to measure symmetry in a configuration, the following definitions are introduced: The *k-neighborhood* ($k \geq 0$) of a processor i consists of the orientation and the input values of the $2k + 1$ processors $i - k, i - k + 1, \ldots, i + k$, relative to the orientation of processor i. Let R be a ring configuration. On a clockwise-oriented ring the k-neighborhood of processor i is represented by the string $I(i - k), \ldots, I(i + k)$; on a general ring it is represented by the string $D(i - k)I(i - k), \ldots, D(i + k)I(i + k)$, if processor i is oriented clockwise ($D(i) = 1$) and by the string $\overline{D(i + k)}I(i + k), \ldots, \overline{D(i - k)}I(i - k)$ if processor i is oriented counterclockwise.

For given ring configuration R and k-neighborhood σ let $g(R, \sigma)$ be the number of processors that have the k-neighborhood σ. $SI(R, k)$, the *symmetry index function* of R, is defined to be

$$SI(R, k) = \min g(R, \sigma),$$

where the minimum is taken over all σ's that are k-neighborhoods of some processor in R. That is, $SI(R, k)$ is the minimum positive number of occurrences of any k-neighborhood in R. This function measures the amount of symmetry in the configuration. If some input value occurs at a unique processor, then $SI(R, k) = 1$ for each k. At the other extreme, if all processors have the same initial state and orientation, then $SI(R, k) = n$ for each k.

The *symmetry index function* $SI(R_1, \ldots, R_j, k)$ of a set of configurations $R_1, \ldots, R_j$ is defined to be the minimum positive total number of occurrences of any k-neighborhood in all the configurations $R_1, \ldots, R_j$; that is,

$$SI(R_1, \ldots, R_j, k) = \min \sum_{i=1}^{j} g(R_i, \sigma),$$

where the minimum is taken over all σ's that are k-neighborhoods of a processor in some configuration R_i.

Let $\omega = I(1), \ldots, I(n)$ be an input in a clockwise-oriented ring; let $\sigma = I(i-k), \ldots, I(i+k)$ be the string representing the k-neighborhood of processor i. There is a one-to-one correspondence between occurrences of that neighborhood in the ring and *cyclic occurrences* of σ in ω; σ *cyclically occurs* in ω if it occurs in some cyclic shift of ω. A similar correspondence exists for nonoriented rings. Let $\omega = D(1)I(1), \ldots, D(n)I(n)$ be the initial configuration of such ring; let $\sigma_1 = D(i-k)I(i-k), \ldots, D(i+k)I(i+k)$ and $\sigma_2 = \overline{D(i+k)}I(i+k), \ldots, \overline{D(i-k)}I(i-k)$. Then there is a one-to-one correspondence between occurrences of the k-neighborhood of processor i in the ring and cyclic occurrences of σ_1 and σ_2 in ω.

3. *Functions that Are Computable on an Anonymous Ring*

We first characterize the functions that can be computed distributively on an anonymous ring. Note that all functions computable on an asynchronous ring can also be computed on a synchronous ring, because synchronous execution is a special case of asynchronous execution. Conversely, we can simulate a synchronous computation on an asynchronous ring using local synchronization: Each processor sends at each cycle synchronization messages to both neighbors; a processor proceeds from the simulation of one cycle to the simulation of the next cycle only after it has received synchronization messages from both neighbors. We, therefore, restrict our attention in this section to the synchronous model. We prove impossibility results for algorithms that solve problems on oriented rings; these impossibility results apply, a fortiori, to general, nonoriented rings.

LEMMA 3.1. *Let R_1 and R_2 be two initial ring configurations, P_1 be a processor in R_1, and P_2 be a processor in R_2 (the rings are not necessarily oriented). Assume that P_1 has the same k-neighborhood in R_1 as P_2 has in R_2. Then the state of P_1 after k cycles of a synchronous algorithm A on input R_1 is identical to the state of P_2 after k cycles of A on input R_2.*

PROOF. By induction on k. □

3.1 KNOWLEDGE OF RING SIZE.

We first remark that knowledge on the size of the ring is necessary for algorithms operating on anonymous rings.

THEOREM 3.2. *Let f: $\{0, 1\}^* \rightarrow \{0, 1\}$ be a nonconstant function. There is no distributed algorithm that computes f correctly on clockwise-oriented rings of arbitrary large size.*

PROOF. Let A be a synchronous algorithm that computes f on clockwise-oriented rings. Consider a ring input I_0 where the answer is 0, and a ring input I_1, where the answer is 1. Assume that A terminates in no more than T cycles on both configurations. Consider now a clockwise-oriented ring with input

$$I = I_0^{(2T+1)} \times I_1^{(2T+1)}.$$

The input I is obtained by concatenating $2T + 1$ copies of I_0, followed by an arbitrary string X, followed by $2T + 1$ copies of I_1. There is a processor in the first segment that has the same T-neighborhood as a corresponding processor in the ring with configuration I_0. This processor will halt and output 0. Similarly, some processor in the second segment will output 1. Hence, the algorithm fails for some input on any oriented ring of length larger than $(2T + 1)(|I_0| + |I_1|)$. □

Any algorithm must have at least a bound on the ring size. For some simple functions the algorithm must "know" exactly the ring size.

THEOREM 3.3. *There is no SUM algorithm that works correctly on two different-sized oriented rings.*

PROOF. Given a SUM algorithm, consider two configurations with all inputs 1, on two oriented rings with different sizes. For any k all k-neighborhoods are identical in both configurations. Thus, the algorithm will output the same answer for both. □

A similar argument proves that an XOR algorithm cannot operate correctly on both even- and odd-size rings.

We henceforth assume that algorithms depend on the ring size n. The algorithms given in this paper depend uniformly on n; the lower bounds are valid for nonuniform algorithms as well.

Even on a ring of fixed size n it is not possible to compute every function distributively. If two processors on rings of size n have the same $\lfloor n/2 \rfloor$-neighborhood, then they have the same k-neighborhood for any k. It follows, by Lemma 3.1, that the output of a synchronous algorithm at a processor is uniquely defined by the $\lfloor n/2 \rfloor$-neighborhood of that processor. This implies that a problem can be solved distributively only if one can associate with each initial configuration R a correct solution O, so that the value of $O(i)$ depends only on the $\lfloor n/2 \rfloor$-neighborhood of i in R. Conversely, we show in the next section that a processor can compute its $\lfloor n/2 \rfloor$-neighborhood, and hence compute any function of that neighborhood. Thus, the last condition is also sufficient.

As a particular case of this general argument we have the following theorem:

THEOREM 3.4. *Let $f: S^n \rightarrow T$. Then:*

(*i*) *There exists an algorithm that computes f on a clockwise-oriented ring of size n iff f is invariant under cyclic shifts of the inputs.*

(*ii*) *There exists an algorithm that computes f on any ring of size n iff f is invariant under cyclic shifts and reversals of the inputs.*

3.2 ORIENTATION. We show in the next theorem that even length rings cannot be oriented distributively. The gist of the argument is that a deterministic algorithm cannot break symmetry in an initial configuration in which half of the processors are oriented in one direction and the other half in the reverse direction, and achieve an asymmetric orientation as output.

THEOREM 3.5. *There is no orientation algorithm that works correctly for even-length rings.*

PROOF. Given a synchronous orientation algorithm that works correctly on a ring of length $2n$, consider the configuration that consists of two oriented half rings (see Figure 1). Formally, we have

$$right(i) = i + 1 \text{ (and } left(i) = i - 1), \qquad \text{for} \quad i = 1, \ldots, n,$$

whereas

$$right(i) = i - 1 \text{ (and } left(i) = i + 1), \qquad \text{for} \quad i = n + 1, \ldots, 2n.$$

The processors i and $2n + 1 - i$ have the same $\lfloor n/2 \rfloor$-neighborhoods so that they halt with the same output. Therefore, the processors i and $2n + 1 - i$ either both

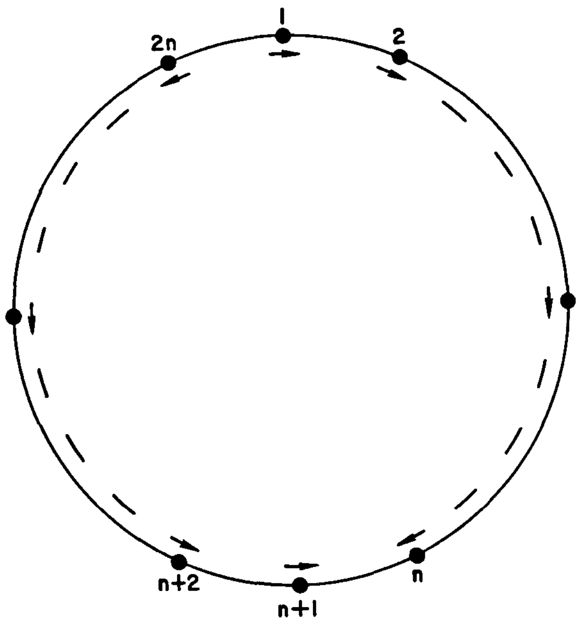

FIGURE 1

switch their orientation or both preserve their original orientation. However, processors i and $2n + 1 - i$ have reverse initial orientations, so that i should switch its orientation iff $2n + 1 - i$ does not switch its orientation. □

The orientation problem can, therefore, be solved only for odd-length rings. An argument similar to that given in the proof of Theorem 3.2 shows the following:

THEOREM 3.6. *There is no algorithm that solves the orientation problem for rings of arbitrarily large size.*

4. *Algorithms*

4.1 THE ASYNCHRONOUS MODEL. We first show that any problem that can be solved on a ring, can be solved using $O(n^2)$ messages; if the inputs are Boolean, then one bit messages are sufficient. This is true even if the ring is not oriented. We present an algorithm that solves the *input distribution* problem. This is the problem of distributing to each processor P the input value and orientation of all processors, relative to the location and orientation of P on the ring. At the end of the computation, each processor holds a string that describes its $\lfloor n/2 \rfloor$-neighborhood. This information is sufficient to compute locally the solution to any problem that can be solved distributively. Hence, input distribution is the "hardest" problem to solve distributively on a ring. By Theorem 3.2 it is necessary to assume that n (the ring size) is known. The algorithm that solves this problem is trivial:

Each processor P first sends, in both directions, a message consisting of its input value and of a bit indicating the port label (e.g., 0 is sent to *left*(P) and 1 is sent to *right*(P)). Next, P forwards the following $\lfloor n/2 \rfloor$-1 messages it receives from *left*(P) to *right*(P), and the following $\lfloor n/2 \rfloor$-1 messages it receives from *right*(P) to *left*(P).

Each processor may reconstitute its $\lfloor n/2 \rfloor$-neighborhood from the messages it receives. Note that the algorithm needs $n(n - 1)$ messages if n is odd. When n is even, $n(n-1)$ messages can be achieved by the following refinement: A processor forwards $n/2 - 1$ messages that were initially sent left and $n/2 - 2$ messages that were initially sent right.

If the ring length is odd, then this input distribution algorithm can be used to orient the ring: processors pick an orientation in accordance with the majority of individual orientations.

4.2 THE SYNCHRONOUS MODEL. A simple adversary argument shows that $\Omega(n)$ messages are needed to compute any nonconstant function. Indeed, consider two input configurations differing at one processor only, such that the corresponding outputs differ. Then each of the remaining $n-1$ processors must receive a message in at least one of the two computations; otherwise, it will have the same output in both computations.

The AND function can be computed with a linear number of messages: A processor with initial state zero sends a message in both directions and halts (in state zero). A processor with initial state one waits for $\lfloor n/2 \rfloor$ cycles. If it receives a message during that period, it then forwards it and halts in state zero. If by cycle $\lfloor n/2 \rfloor$ the processor has not received any message, it halts in state one. The total number of messages sent is $O(n)$.

All algorithms in this section have a similar flavor: They may "deadlock" in symmetric configurations where all processors are waiting for messages and no message is outstanding; the processors detect such situation and break the deadlock by counting the number of cycles since last message arrival. The main use of synchronism is to provide an upper bound on message arrival time.

4.2.1 *Input Distribution Algorithm.* Not every function can be computed in a linear number of messages (see Section 6). We show that $O(n \log n)$ messages are always sufficient to solve a problem on a synchronous ring of size n if a distributed solution exists. An algorithm that solves the input distribution problem using $O(n \log n)$ messages in the worst case is presented in this section. We describe the algorithm assuming the ring is oriented; later we extend the algorithm to non-oriented rings.

In a ring where processors have distinct labels, a leader can be selected using $O(n \log n)$ messages [5, 8, 12]. The leader is selected to be the processor with maximum label. The selection algorithm processes by rounds: Initially each processor is *active.* At each round the remaining active candidates that are *neighbors* exchange and compare labels, whereby a fixed proportion of them are disqualified, and become *passive* (two active processors are neighbors if there is no other active processor on an arc connecting them). After $O(\log n)$ rounds, each requiring $O(n)$ messages, only one leader remains. Once a leader is selected, it can collect the inputs on the ring and rotate the information to the remaining processors.

Our input distribution algorithm uses a similar approach. Since labels do not preexist, we create them during the computation: We label an active processor P with the string of inputs of the processors on the segment starting from the previous active processor at the left of P and ending at P.

In the first phase of a round the active processors use these labels to perform comparisons. Each active processor sends its label in both directions. Passive processors forward these messages. An active processor is disqualified if it receives a label that is greater than its label (in lexicographic order) or if both labels it receives are equal to its label. This phase takes n cycles and $2n$ messages. If a processor wins in this phase, then at least one of its neighbors loses. It follows that at least one-third of the active processors are disqualified in this phase, and the total number of rounds is at most $\log_{1.5} n$.

In the second phase of a round the remaining active processors collect their new labels: Each active processor sends an empty message to its right. A processor that receives such a message appends to it its own input and, if it is passive, forwards the augmented message. After n cycles and n messages, each active processor has computed its label. At each phase of each round, each processor receives at least one message.

There is, however, no guarantee that the labels used for leader election are distinct. The algorithm may deadlock in a symmetric configuration where all active processors have the same labels, in which case no active processor is left at the next round. Since the algorithm is synchronous, the processors can detect that such an event occurred by waiting a sufficiently long time: A processor detects a deadlock situation if it does not receive any message during the n cycles of the second phase of a round. At this point we are in a situation in which the initial configuration is periodic, and each active processor holds a copy of that period. Since the ring size is known, each processor can reconstruct the entire initial configuration from this period.

Termination when only one active processor is left is handled in the same way: The processor competes against itself and is eliminated at the next round.

We give a formal description of the algorithm in Figure 2. The symbol & denotes string concatenation; cyclic_shift is the function that cyclically shifts characters in a string one position to the left. The algorithm uses at most $n(3 \log_{1.5} n + 1)$ messages and runs for $n(2 \log_{1.5} n + 1)$ cycles.

It is easy to modify the last algorithm so as to use only one-sided communication. Thus, any problem that can be solved on a unidirectional ring can be solved synchronously in $O(n \log n)$ messages.

The messages used in the last algorithm may require a linear number of bits; it is possible to replace them with "zero content" messages, using time to encode information. If there are k different types of messages, then we replace each cycle by k subcycles and represent a message of type i sent at cycle t by an empty message sent at cycle $k(t - 1) + i$.

4.2.2 *Orientation Algorithm.* In order to apply the previous algorithm to arbitrary rings, we first have to orient the ring. By Theorem 3.5 this is not possible, in general, if the ring has even length. However, a weaker result is sufficient for our purposes. A ring is said to be *quasi-oriented* if either the ring is oriented, or the ring orientation alternates, that is, successive processors on the ring have opposite orientations. In this section we present an algorithm that quasi-orients an arbitrary ring, using $O(n \log n)$ bit messages and $O(n \log n)$ cycles.

It is easy to modify the input distribution algorithm so as to work on a ring with alternate orientation: One runs two computations simultaneously, one for each direction. Processors participate in one computation and forward messages of the other computation. At the end, neighbors exchange information. Hence, the quasi-orientation algorithm and the input distribution algorithm for oriented rings can be combined into an input distribution algorithm that works for rings with arbitrary orientation. This algorithm can also distribute the initial relative orientation of each processor.

Note that a quasi-oriented ring of odd length is oriented; a quasi-orientation algorithm orients rings of odd length.

```
active := true;
label := input;
done := false;
repeat      { do one round }

    { first phase - elimination }
    if active
        then begin
            send label to left and right;
            wait(n-1);
            winner := label $\geq$ each message received
                and label $>$ at least one message received
            end
        else    { passive processors }
            for i := 1 to n do forward message;

    { second phase - label creation }
    if active and winner
        then begin
            send nil to right;
            wait(n-1);
            label := (message received)&input
            end
        else begin
            done := true;
            for i := 1 to n do
                if received M
                    then begin
                        active := false;
                        done := false;
                        send M&input to right
                        end
            end
until done;

{ broadcast result}
if active
    then begin
        send label to right;
        halt
        end
    else for i :=1 to n do
        if received M
            then begin
                label := cyclic_shift(M);
                send label to right;
                halt
                end
```

FIG. 2. Input distribution algorithm.

The quasi-orientation algorithm is similar to the input distribution algorithm. It proceeds by rounds, each taking a linear number of cycles and messages and each disqualifying a constant fraction of the active processors. When the elimination procedure "deadlocks," the pattern of active processors is such that the ring can be quasi-oriented in one more pass.

An active processor P is an *endpoint* if the first active processor to the left of P is oriented differently from P. Endpoints define *segments.* A segment consists of consecutive processors on the ring, starting with an endpoint that is oriented clockwise and ending with the next endpoint (that is oriented counterclockwise). Each active processor is contained in some segment (see Figure 3). Each round has two phases. The first phase determines which active processors are endpoints. At

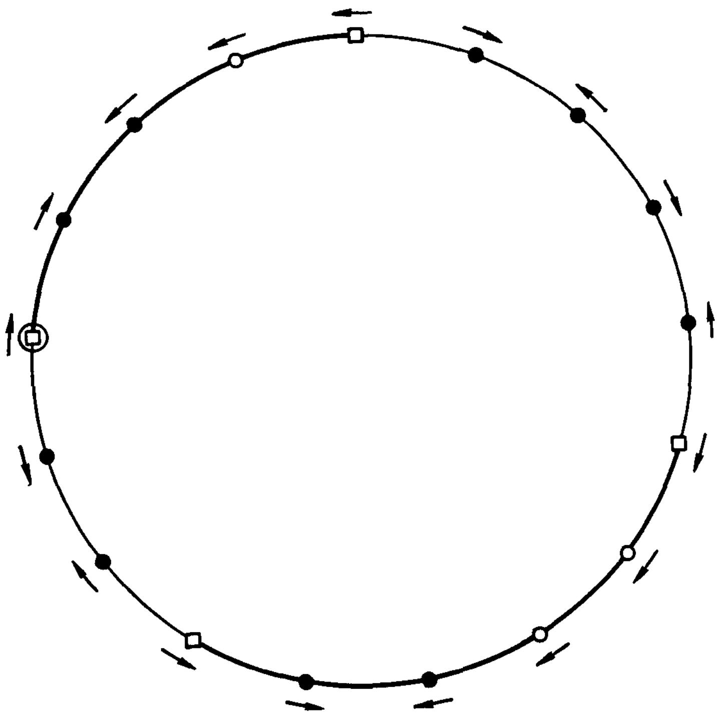

FIG. 3. ●, passive processor; ○, active processor; □, endpoint; ◎, processor active at next round.

the start of this phase each active processor sends distinct messages to its left and to its right. An active processor is an endpoint if it receives from its left a message sent to the left. This phase uses n cycles and at most $2n$ messages.

In the second phase eliminations are made. One endpoint is selected from each odd-length segment to stay active for the next round, and the remaining active processors become passive. At the start of this phase endpoints send messages to their right. A processor that receives two messages simultaneously (this happens only in the middle of a segment of odd length) sends a reply to one of the endpoints of its segment. This endpoint stays active.

It is easy to see by induction that the number of passive processors between two active processors is even at the end of each round. This is true initially, when all processors are active. Assume it holds true at the start of a round. Then each even- (odd-) length segment contains an even (odd) number of active processors. It follows that the number of active processors eliminated in each segment is even, so that the claim holds at the end of the round. The same argument also implies that the number of active processors decreases at least by a factor of 3 at each round.

The second phase uses at most $1.5n$ messages and requires n cycles. The computation will halt after at most $1 + \log_3 n$ rounds.

The algorithm was designed so that each processor receives at least one message at each round, except the last one (this is the reason messages are sent in both directions in the first phase of a round). Hence a processor can detect, by waiting a sufficiently long time, whether the computation has halted at the previous round.

The computation may stop for two reasons:

(1) No endpoints were found at the first phase of a round. Then all the active processors left from the previous round have the same orientation; they can orient the ring.

(2) All segments had even length; hence no new active processor was selected at the second phase of a round. Then any two neighboring endpoints have reverse orientation and are at odd distance one from the other; all even processors that are endpoints have the same orientation, and so do all odd processors that are endpoints. Hence, the even processors can be oriented consistently and so can the odd processors, resulting in an alternate orientation.

The orientation algorithm is described in Figure 4. It uses at most $n(2 \log_3 n + 4)$ cycles and at most $3.5n(\log_3 n + 1)$ messages.

4.2.3 *Start Synchronization.* Algorithms in this section were written under the assumption that processors start simultaneously. We now drop this assumption. We assume that the processors were all originally *idle.* A processor awakes either spontaneously or when receiving a message. The transitions are henceforth deterministic.

We show that the *start synchronization* problem can be solved in $O(n \log n)$ messages: We give an algorithm such that, if each processor starts running the algorithm at a (possibly) different time, then $O(n \log n)$ cycles and messages after the first processor has started, all processors will halt simultaneously. By prefixing the synchronization algorithm to an algorithm that assumes simultaneous start, we obtain an algorithm that solves the same problem but does not require simultaneous start.

The synchronization algorithm will synchronize all processors to the time of the earliest starting processor(s). It uses the same basic framework as our input collection algorithm. We run a leader-choosing algorithm, which chooses the earliest running processor. We assume that each processor keeps a count of the number of cycles that have lapsed since it awoke. An active processor is a local maximum if its count is ahead of the count of its neighbors and strictly ahead of the count of at least one neighbor. At most, two-thirds of the active processors survive after each round. If no local maximum is found, then all active processors have the same count and, thereby, are synchronized.

A difficulty arises when neighboring active processors exchange their counts: While the message containing the count travels, the value of the count changes. This can be remedied by having each passive processor increase the count it forwards by one.

We give a formal description of this algorithm in Figure 5.

Let P be a processor that wakes up earliest. Then all processors are alive when P count is equal to $\lfloor n/2 \rfloor$. This implies that the counts of distinct processors are apart by at most $\lfloor n/2 \rfloor$. Let round k consist of the cycles in which processors forward messages that were initiated by active processors when their count was equal to $2kn$. Since all messages reach their destination in n cycles at most, the last remark implies that distinct rounds do not overlap.

When a round ends, all processors that received a message from a local maximum are synchronized with it. If one processor receives a message during a round, then all do. If no processor received a message during a round, then no local maximum was detected at the previous round, and all processors are synchronized. This implies the correctness of the algorithm.

```
done := false
active := true;
repeat    { do one round }

    { first phase - select endpoints }
    if active
        then begin
            send message LEFT to left;
            send message RIGHT to right;
            wait(n-1);
            if not received LEFT from left
                then begin
                    active := false;
                    marked := true
                    end
            end
        else begin    { passive }
            done := true;
            for i:=1 to n do
                if received message then
                    begin
                    done := false;
                    marked := false;
                    forward message
                    end
            end;

      { second phase - eliminate }
    if active
        then begin
            send message 0 to right;
            wait(n-1);
            if not received 1
                then begin
                    active := false;
                    marked := true
                    end
            end
        else    { passive }
            for i := 1 to n do
                if received message
                    then begin
                        marked := false;
                        if received 0 from left and right
                            then send 1 to right
                            else if message = 1 or first message received
                                then forward message
                        end
until done;

   { active processors orient processors of same parity }
if marked
    then begin
        send 0 to right;
        message_count :=1
        end
    else message_count := 0;
repeat forever
    if received message M
        then begin
            message_count := message_count+1;
            if M=1 and M received from right
                then switch orientation;
            forward $\overline{M}$;
            if message_count = 2 then halt
end
```

FIG. 4. Orientation algorithm.

```
{ Initially all processors have count = -1
  and are not active }

if wakes up spontaneously
   then begin
      count := 0;
      active := true;
      message_count := 0;
      send count to left and to right
      end;
repeat forever
   begin
   count := count+1;
   if count mod 2n = 0
      then if no message was received in last 2n cycles
         then halt
      else if active
         then send count to left and to right;
   if  count mod 2n ≠ 0
      then if received message M
         then if not active
            then begin
               forward M+1;
               count := max (M+1,count)
               end
            else { active }
               begin
               message_count := message_count+1;
               count := max (M+1, count);
               if message_count = 2
                  then if not local maximum
                     then active := false
                     else message_count := 0;
               end
   end
```

FIG. 5. Synchronization algorithm.

During a round at most $2n$ messages are sent. The number of active processors decreases at each round by one-third at least, so that the total number of rounds is at most $1 + \log_{1.5}n$. The total number of messages sent is at most $2n(1 + \log_{1.5}n)$.

4.2.4 *Bit Complexity of Start Synchronization.* In the synchronization algorithm it is possible to replace arbitrary messages by bit messages, with a constant-factor increase in time and message complexity. Note that each processor knows, upon receiving a message, to which round it pertains. This determines the clock count of the processor that originated the message at the time the message was sent. The only required information is the time that has elapsed from the issuing of the message to its reception, that is, the distance separating the sender from the receiver. This distance can be computed by sending two messages: first a message traveling at speed 1 and next a message traveling at speed $\frac{1}{2}$: The first message is forwarded at each cycle, whereas the second message is delayed one cycle by each forwarding processor. The distance between sender and receiver equals the difference between the reception time of these two messages.

When this protocol is used, the number of cycles required for a communication between two active processors is bounded by $2n$. We modify the algorithm so that $3n$ cycles are dedicated to each round (an active processor sends a message only when its count is equal to $3kn$); active processors send messages using the method described above, and passive processors merely forward messages. The algorithm will require no more than $3n\log_{1.5}n$ cycles and $4n\log_{1.5}n$ messages.

5. *Lower Bounds for the Asynchronous Model*

5.1 THE GENERAL THEOREM. In this section we give $O(n^2)$ asynchronous lower bounds for simple functions such as AND, as well as for orienting the ring. A general lower bound technique is first introduced: We define pairs of initial configurations with certain parameterized properties and then show that the existence of such a pair for a particular problem implies that a certain number of messages (which depends on the parameters of the pair) must be sent. Note that message complexity always means the number of messages sent in the worst case over all possible asynchronous computations.

The same method works for the synchronous case (next section), but a stronger property of the pair is required and the proofs are more involved.

Definition. Let α be a constant and $\beta(\cdot)$ be a function. Two initial configurations R_1 and R_2 of length n are an α, β *fooling pair* for a distributed algorithm if the following conditions hold:

(5a) For any output configurations O_1 and O_2 obtained by running the algorithm on initial configurations R_1 and R_2, respectively, there exist two processors, P_1 and P_2, such that P_1 has the same α-neighborhood in R_1 as P_2 has in R_2 but $O_1(P_1) \neq O_2(P_2)$.

(5b) The symmetry index function of R_1 is bounded by β, that is,

$$SI(R_1, k) \geq \beta(k),$$

for $0 \leq k \leq \alpha$ (see definition at the end of Section 2).

For example, clockwise-oriented rings with inputs $I_1 = 1^n$ and $I_2 = 1^{n-1}0$ are a fooling pair for any algorithm that computes the AND function, with $\alpha = \lfloor n/2 \rfloor - 1$ and $\beta(k) = n$ for every $1 \leq k \leq n$. Indeed, processor $\lfloor n/2 \rfloor$ has the same α-neighborhood in both configurations; yet its output in the first configuration is 1 and in the second 0, and all processors in the first configuration have the same k-neighborhood for any k.

THEOREM 5.1. *Let R_1 and R_2 be input configurations of length n that are a fooling pair with parameters α and β for an asynchronous algorithm A. Then A uses at least $\sum_{k=0}^{\alpha} \beta(k)$ messages on R_1 in the worst case.*

PROOF. In the asynchronous model an adversary can manipulate the delays of the communication channels. We use an adversary that "synchronizes" the computation, thus keeping the ring configuration as symmetric as possible. The adversary schedules arrivals of messages in successive cycles so that all messages sent at cycle i are received at cycle $i + 1$. The schedule is formally defined as follows: All processors execute their initial state transition at cycle 0; no message is received at this cycle.

At cycle $i + 1$ each processor that has not halted receives successively all messages sent at cycle i by its left neighbor and next receives successively all messages sent at cycle i by its right neighbor, in the order in which they were sent.

As in Lemma 3.1, a simple induction shows that the state of a processor at the end of k cycles depends only on the k-neighborhood of the processor.

Consider the computations of A on R_1 and R_2 using the above adversary. Let T be the first cycle in which no message is sent in the first computation. No message arrival, and hence no state transition, occurs at cycle $T + 1$ and at any subsequent cycle. Hence, the computation must have terminated at the end of cycle T; at least one processor has sent a message at each cycle t of the computation on R_1, for $0 \leq t < T$.

Look at processors P_1 and P_2. Assume $T \leq \alpha$. Since P_1 and P_2 have the same T-neighborhood, P_1 and P_2 halt with the same output in both computations; a contradiction. Hence $T > \alpha$.

Assume processor Q sends a message at cycle k of the computation on R_1. Then any processor with the same k-neighborhood also sends the same message. By condition (5b), there are $\beta(k)$ such processors. Hence, at least $\beta(k)$ messages are sent at cycle k, for $0 \leq k \leq \alpha$, and the claim follows. □

5.2 Lower Bound Examples

5.2.1 *Computing AND.* As mentioned before, the oriented rings with input configurations $I_1 = 1^n$ and $I_2 = 1^{n-1}0$ are a fooling pair for any algorithm that correctly computes the AND function, with $\alpha = \lfloor n/2 \rfloor - 1$ and $\beta(k) = n$, for any k. Theorem 5.1 implies that any algorithm computing AND will send at least $\sum_{i=0}^{\lfloor n/2 \rfloor - 1} n = n \cdot \lfloor n/2 \rfloor$ messages on I_1 in the worst case.

By distinguishing cycles in which transmissions occur in both directions from cycles in which transmissions occur in one direction only, one can improve the lower bound for AND to $n(n-1)$ messages, which is tight. Since AND computes the minimum of a set of $\{0, 1\}$ values, this implies the following:

Corollary 5.2. *Any asynchronous algorithm for computing the minimum of all inputs on a ring requires at least $n(n-1)$ messages in the worst case, when the inputs are not necessarily distinct.*

(This result was proved independently by Itai [9].) This is an interesting contrast to the case in which all inputs are distinct, and the minimum can be computed in $O(n \log n)$ messages [4, 5, 8, 12].

Theorem 5.1 can be used to obtain an $\Omega(n^2)$ lower bounds on the number of messages required to compute any function f: $\{0, 1\}^n \rightarrow \{0, 1\}$ with the property that $f(0, \ldots, 0) \neq f(1, \ldots, 1)$. Indeed, if $f(1^n) \neq f(0^{\lceil n/2 \rceil}1^{\lfloor n/2 \rfloor})$, then 1^n and $0^{\lceil n/2 \rceil}1^{\lfloor n/2 \rfloor}$ are a fooling pair for any algorithm that computes f, with $\alpha = \lfloor (n-2)/4 \rfloor$ and $\beta(\cdot) = n$; if $f(0^n) \neq f(0^{\lceil n/2 \rceil}1^{\lfloor n/2 \rfloor})$, then 0^n and $0^{\lceil n/2 \rceil}1^{\lfloor n/2 \rfloor}$ are a fooling pair with the same parameters. As $f(0^n) \neq f(1^n)$, one of these two alternatives must hold.

5.2.2 *Orienting the Ring.* We have shown that processors on a ring need $\Omega(n^2)$ messages to agree on an identical Boolean output whenever the agreement problem is nontrivial (e.g., whenever processors are not allowed to agree on the same value for the all-zeros configuration as for the all-ones configuration). We now show that $\Omega(n^2)$ messages are required for the processors to agree on a consistent orientation. To prove this lower bound, we use Theorem 5.1. Remember that the notion of neighborhood includes the relative orientations of processors.

Theorem 5.3. *Any asynchronous algorithm for orienting a ring requires at least $\Omega(n^2)$ messages in the worst case.*

Proof. We look at two initial configurations of odd length $n = 2m + 1$ (even-length configurations cannot be oriented by Theorem 3.5). The first configuration R_1 is the clockwise-oriented ring. In the second configuration R_2, processors $1, \ldots, m$ are clockwise oriented ($right(i) = i + 1$, for $i = 1, \ldots, m$) and the processors $m + 1, \ldots, 2m + 1$ have the reverse orientation ($right(i) = i + 1$, for $i = m + 1, \ldots, 2m + 1$) (see Figure 6).

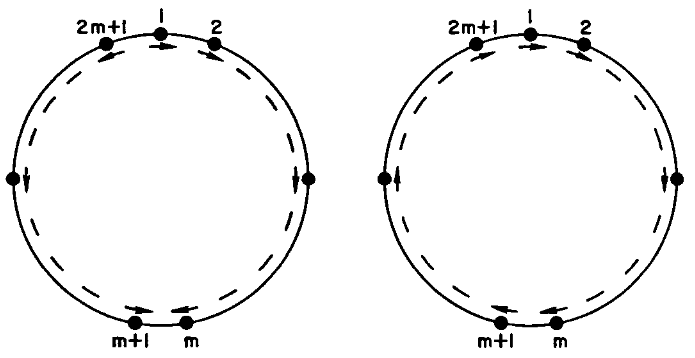

FIGURE 6

Processors $\lceil n/4 \rceil$ and $\lceil 3n/4 \rceil$ have opposite initial orientations in R_2. The algorithm always produces an oriented ring, and thus these processors always halt with distinct outputs in R_2. Hence one of them has an output that is distinct from the output of processor 1 in configuration R_1. Moreover, all three processors have the same $\alpha = \lfloor (n-2)/4 \rfloor$-neighborhood. Hence condition (5a) is satisfied for that particular value of α. Clearly, R_1 satisfies condition (5b) with $\beta(r) = n$ for all r. Thus, R_1 and R_2 are a fooling pair for the orienting algorithm, with $\alpha = \lfloor (n-2)/4 \rfloor$ and $\beta(r) = n$.

We apply Theorem 5.1 and obtain an $n \lfloor (n+2)/4 \rfloor$ lower bound for orientation. □

A more refined analysis can be applied in order to raise the last lower bound by factor of 2. An $\Omega(n^2)$ lower bound quasi-orientation (on even rings) can be obtained in a similar manner.

5.2.3 *Random Functions.* Functions with linear or even subquadratic message complexity are rare: "almost all" functions have quadratic complexity.

THEOREM 5.4. *The probability that a random computable Boolean function on n variables has message complexity $\leq n^2/4$ on oriented rings is less than* $2^{1-(2^{n/2}/n)}$.

PROOF. Define two input configurations to be equivalent if one is a cyclic shift of the other. This partitions the input configurations into equivalence classes. Each class has size at most n, and every computable function is invariant on each class (Theorem 3.4).

Let I be a configuration with $n/2$ contiguous ones. It is easy to see, using Theorem 5.1, that, if $f(I) \neq f(1^n)$, then at least $n^2/4$ messages are sent on input 1^n by any algorithm that computes f.

Let s be the number of equivalence classes that contain a configuration with $n/2$ contiguous 1's. If a computable Boolean function can be computed in fewer than $n^2/4$ messages, then it has the same value on all these classes. This happens with probability $\leq 2^{1-s}$. There are $2^{n/2}$ strings starting with $n/2$ ones; each equivalence class contains at most n of these strings. Hence $s \geq (2^{n/2}/n)$, and the bound follows. □

6. *Lower Bounds for Synchronous Rings*

6.1 THE GENERAL LOWER BOUND TECHNIQUE. The synchronous algorithms presented in this paper and the election algorithm of [7] exhibit a powerful feature of synchronous systems, in which time can convey information. In Section 4 we traded message arrival time for message content. Moreover, even the nonarrival of a message was in itself informative. However, observe the following intuitive fact: A cycle in which no message is received advances the computation only if there is some other computation in which a message is received at this cycle. The lower bound argument presented in this section uses this observation.

Consider the computations of an algorithm A running on two initial configurations R_1 and R_2. A cycle is *active* if some message is sent in at least one of the two computations at this cycle. We refine Lemma 3.1 to apply to active cycles.

LEMMA 6.1. *Let A be a synchronous algorithm running on two configurations R_1 and R_2. Let P_1 and P_2 be two processors in R_1 and R_2, respectively, with the same k-neighborhood. Then P_1 and P_2 are in the same state after k active cycles of the computations of A on the respective input configurations.*

PROOF. A simple induction on k. □

As in the previous section we want to build a fooling pair with many symmetries. Again we assume that we have two configurations, with two processors having the same (large) neighborhood, that should reach different output states. However, we cannot settle for only one symmetric configuration, as in the asynchronous case: The computation may proceed, with few messages, on the asymmetric configuration, and processors will cheaply detect the symmetric configuration by noticing that no messages are generated. (Think about the asynchronous lower bound and the synchronous algorithm for AND.)

The definition of a fooling pair for the synchronous model is modified to ensure that no neighborhood occurs infrequently in both configurations.

Definition. Two initial configurations R_1 and R_2 of length n are an α, β *fooling pair* for a synchronous algorithm A if the following conditions hold:

(6a) There exist two processors, P_1 in R_1 and P_2 in R_2, such that P_1 and P_2 have the same α-neighborhood and the output of P_1 in the computation of A on R_1 is different from the output of P_2 in the computation of A on R_2.

(6b) The symmetry index function of the set of configurations R_1, R_2 is bounded by β, that is, $SI(R_1, R_2, k) \geq \beta(k)$ for any $0 \leq k \leq \alpha$.

THEOREM 6.2. *Let R_1 and R_2 be an α, β fooling pair for a synchronous algorithm A. Then A sends at least $\frac{1}{2}\sum_{k=0}^{\alpha}\beta(k)$ messages on one of these two configurations.*

PROOF. Let P_1 and P_2 be the processors defined by property (6a). Let T be the number of active cycles performed by the algorithm on the two configurations. If $T \leq \alpha$, then according to Lemma 6.1, P_1 and P_2 are in the same state at the end of the computation and output the same value; a contradiction. Thus $T > \alpha$.

Let P be a processor that sends a message at the kth active cycle in one of the two computations. By Lemma 6.1, any processor with the same $(k-1)$-neighborhood sends a message too. However, by property (6b) such neighborhood occurs at least $\beta(k-1)$ times in configurations R_1 and R_2. At least this number of messages is sent in both computations at the kth active cycle. The total number of messages sent in both computations is at least equal to $\sum_{k=0}^{\alpha}\beta(k)$. It follows that at least $\frac{1}{2}\sum_{k=0}^{\alpha}\beta(k)$ messages are sent in one of the two computations. □

The two configurations of a fooling pair need not be distinct; a fooling pair may be defined by two copies of one configuration R that fulfills (6b) (i.e., $SI(R, R, k) = 2SI(R, k) \geq \beta(k)$) and two distinct processors in that configuration that fulfill (6a). Theorem 6.2 is still valid in such setting.

6.2 STRING CONSTRUCTION. Unlike the asynchronous case, the construction of a synchronous fooling pair is a complicated task, and a special technique is required. We build symmetric configurations by repeated applications of a word homomorphism.

The properties of languages obtained by repeated applications of word homomorphisms (D0L systems) have been studied by many authors (see [13–15]). Our results on the symmetry index function of such strings are most directly related to the results in [6].

Let Σ be a finite alphabet. Let h be a homomorphism from Σ to Σ^*. We restrict ourselves to the case $\Sigma = \{0, 1\}$; however, the results can be generalized. We assume that h fulfills the following two conditions:

(6c) Every word of length 2 occurs both in $h^c(0)$ and in $h^c(1)$, for some constant c.

(6d) The homomorphism h is *uniform*; that is, $|h(0)| = |h(1)| = d$, with $d \geq 2$.

Note that if h is uniform then $|h^k(\omega)| = d^k|\omega|$.

We show that all strings ω that are obtained by many repeated applications of h contain the same (short) substrings, and that each (short enough) substring σ occurs $\Theta(|\omega|/|\sigma|)$ times in ω, if it occurs at all. This is formally stated in the next theorem.

THEOREM 6.3. *Let h be a homomorphism that fulfills conditions* (6c) *and* (6d). *Then there exist positive constants a and b* ($a = 1/d^c$ *and* $b = 1/d^{c+1}$) *such that the following holds*:

If σ occurs cyclically in $\omega = h^k(\rho)$, and $|\sigma| \leq a|\omega|/|\rho|$, then σ occurs at least $b|\omega'|/|\sigma|$ times in any string $\omega' = h^k(\rho')$.

We first prove the claim for substrings of length 2.

LEMMA 6.4. *Let h be a homomorphism that fulfills conditions* (6c) *and* (6d). *Then any string of length* 2 *occurs at least d^{k-c} times in $h^k(0)$ and in $h^k(1)$, for $k \geq c$.*

PROOF. Let $h^{k-c}(0) = \epsilon_1 \cdots \epsilon_n$. Then $n = d^{k-c}$ and $h^k(0) = h^c(\epsilon_1 \cdots \epsilon_n) = h^c(\epsilon_1), \ldots, h^c(\epsilon_n)$. The claim follows since any string of length 2 occurs at least once in each $h^c(\epsilon_i)$. □

In order to extend the lemma to strings of arbitrary length we prove that any (short enough) string that occurs in $h^k(\rho)$ is obtained by repeated applications of h on a string of length at most 2.

LEMMA 6.5. *Let h be a uniform homomorphism and let σ be a string of length $|\sigma| \leq |\omega|/|\rho|$ that occurs cyclically in $\omega = h^k(\rho)$. Then σ is a substring of $h^i(\pi)$, where $|\pi| \leq 2$, and $i = \lceil \log_d|\sigma| \rceil$.*

PROOF. Note that $|\omega|/|\rho| = d^k$; hence $k \geq i$. Since σ occurs cyclically in $h^k(\rho)$, it occurs (noncyclically) in $h^k(\rho\rho) = h^i(h^{k-i}(\rho\rho))$. Let π be a minimum length substring of $h^{k-i}(\rho\rho)$, such that σ occurs in $h^i(\pi)$. Because $|h^i(0)| = |h^i(1)| = d^i \geq |\sigma|$, $|\pi| \leq 2$. □

PROOF OF THEOREM 6.3. Assume that σ occurs cyclically in $\omega = h^k(\rho)$ and that $|\sigma| \leq |\omega|/(d^c|\rho|)$. According to the previous lemma σ occurs in $h^i(\pi)$, where $|\pi| \leq 2$ and $i = \lceil \log_d|\sigma| \rceil$. We have $|\sigma| \leq d^{k-c}$; hence $k - c - i \geq 0$. By Lemma 6.4 π occurs at least $d^{k-i-c}|\rho'|$ times in $h^{k-i}(\rho')$. Hence σ occurs at least $d^{k-i-c}|\rho'|$ times in $h^i(h^{k-i}(\rho')) = \omega'$. But $d^{k-i-c}|\rho'| \geq |\omega'|/(d^{c+1}|\sigma|)$. Thus σ occurs at least $|\omega'|/(d^{c+1}|\sigma|)$ times in ω'. □

COROLLARY 6.6. *Let ρ_1 and ρ_2 be strings with $|\rho_1| = |\rho_2| = s$, and let $I_i = h^k(\rho_i)$, $i = 1, 2$, where h is a homomorphism that fulfills conditions* (6c) *and* (6d). *Let f be a function such that $f(I_1) \neq f(I_2)$. Then any synchronous algorithm that computes f on oriented rings of length $n = sd^k$ requires in the worst case*

$$\frac{n}{2d^{c+1}} \ln\left(\frac{n}{sd^c}\right)$$

messages.

PROOF. Let R_1 and R_2 be clockwise oriented rings with input configurations I_1 and I_2, respectively. If A is an algorithm that computes f, then the output of a processor in R_1 is distinct from the output of a processor in R_2 when A is run. A k-neighborhood in R_i corresponds to a string of length $2k + 1$ in I_i. Hence any k-neighborhood that occurs in R_1 or in R_2 occurs at least $n/d^{c+1}(2k + 1)$ times both in R_1 and in R_2, whenever $2k + 1 \leq n/sd^c$. This implies that R_1 and R_2 are a fooling pair for A with $2\alpha + 1 = n/sd^c$ and $\beta(k) = 2n/d^{c+1}(2k + 1)$. By Theorem 6.2 algorithm A will send on one of these two configurations at least

$$\frac{1}{2} \sum_{2k+1 \leq n/sd^c} \frac{2n}{d^{c+1}(2k+1)} \geq \frac{n}{2d^{c+1}} \ln\left(\frac{n}{sd^c}\right)$$

messages. □

6.3 LOWER BOUND EXAMPLES

6.3.1 *Computing XOR.* To apply Corollary 6.6 to XOR, we construct a homomorphism h that fulfills conditions (6c) and (6d) so that the XOR function obtains different values on $h^k(0)$ and on $h^k(1)$. Consider the following homomorphism:

$$h(0) \to 011, \qquad h(1) \to 100.$$

Any string of length 2 occurs in $h^2(0)$ and in $h^2(1)$. Note that $h^k(1) = \overline{h^k(0)}$ and $|h^k(0)| = 3^k = |h^k(1)|$. The string $h^k(0)$ has an even number of 1's, whereas $h^k(1)$ has an odd number of 1's. Thus, XOR obtain different values on these two configurations. This implies, by Corollary 6.6, that computing XOR requires at least $(n/54)\ln(n/9)$ messages.

6.3.2 *Orienting the Ring.* Let h be the homomorphism defined by

$$h(0) \to 011, \qquad h(1) \to 001.$$

Any string of length 2 occurs in $h^2(0)$ and in $h^2(1)$. Let $D = D(1), \ldots, D(n) = h^k(0)$ be an orientation for a ring of odd length $n = 3^k$. We have $h(0) = \overline{h(1)^R}$ and, by induction, $h^k(0) = \overline{h^k(1)^R}$; that is, $h^k(0)$ is obtained from $h^k(1)$ by complementing each bit and reversing the string. It follows that

$$h^k(0) = h^{k-1}(0)h^{k-1}(1)h^{k-1}(1) = h^{k-1}(0)\overline{h^{k-1}(0)^R}\,\overline{h^{k-1}(0)^R}.$$

This implies that processor $\lceil n/6 \rceil$ (which is in the middle of the first third of the ring) and processor $\lceil n/2 \rceil$ (which is in the middle of the second third of the ring) have identical ($\lceil n/6 \rceil - 1$)-neighborhoods in the ring defined by D, but opposite orientations; their output in a computation that orients the ring is distinct.

Let σ be a string of length $2k + 1 \leq n/9$ that occurs cyclically in D. By Theorem 6.3, σ occurs at least $n/(27 \, |\sigma|)$ times in D. It is easy to show, using Lemma 6.5, that $\bar{\sigma}^R$ also occurs in D and therefore occurs at least $n/(27 \, |\sigma|)$ times. Both strings define identical k-neighborhoods in D. Thus, each k-neighborhood occurs at least $2n/(27(2k + 1))$ times, whenever $2k + 1 \leq n/9$. The ring with orientation D is a fooling pair for a synchronous orienting algorithm, with $2\alpha + 1 = n/9$ and $\beta(k) = 4n/(27(2k + 1))$. Using Theorem 6.2, we get a lower bound of

$$\frac{1}{2} \sum_{2k+1 \leq n/9} \frac{4n}{27(2k + 1)} \geq \frac{n}{27} \ln \frac{n}{9},$$

on the number of required messages.

6.3.3 *Start Synchronization.* When proving lower bound on synchronization, an adversary controls the starting time of the processors. When a processor awakes, it can send a message to its neighbor, and the neighbor then awakes; hence, an adversary is allowed to schedule a processor to wake up at most 1 time cycle apart from its neighbors.

With a $\{0, 1\}$-string $\omega = \epsilon_1, \ldots, \epsilon_n$, we associate a starting configuration R_ω of size n in the following way: A dummy processor 0 starts at cycle 0; If processor $i - 1$ starts at cycle t_{i-1}, then processor i starts at cycle $t_{i-1} + 1$ if $\epsilon_i = 1$, and at cycle $t_{i-1} - 1$ if $\epsilon_i = 0$, for $1 \leq i \leq n$. The resulting assignment of starting times is valid for a ring provided that the starting cycle of the first and last processor differ at most by 1.

Consider the previously defined homomorphism:

$$h(0) \rightarrow 011, \qquad h(1) \rightarrow 100.$$

Let $m = 3^k$ and $n = 4m$. Let $\sigma_0 = h^k(0)$ and $\sigma_1 = h^k(1)$. Note that $h^k(0) = \overline{h^k(1)}$. Consider the following string:

$$\omega = \sigma_0\sigma_0\sigma_1\sigma_1 = h^k(0011).$$

We associate with ω a starting configuration R_ω, as previously described. The number of ones is not equal to the number of zeros in σ_0; hence $t_m \neq 0$. As $\sigma_1 = \bar{\sigma}_0$, the string $\sigma_0\sigma_0\sigma_1\sigma_1$ has equal number of ones and zeros; this implies that $t_n = 0$. Hence this is a legal scheduling.

Let A be a synchronization algorithm. We assume without loss of generality, that the output of a processor is the number of cycles from its wakeup time. Processors $\lceil m/2 \rceil$ and $\lceil 3m/2 \rceil$ start at different cycles and hence must have distinct outputs. These processors have the same $\lfloor m/2 \rfloor$-neighborhood. Each k-neighborhood occurs at least $4m/27(2k + 1)$ times, whenever $2k + 1 \leq m/9$. This implies an $(n/54)\ln(n/36)$ lower bound on the number of messages sent by A.

6.3.4 *Random Synchronous Functions.* In the previous sections we have seen that some functions, such as AND, can be computed with linear number of messages, whereas other functions, like XOR, have message complexity of $\Theta(n \log n)$ in the worst case. It turns out that the behavior of XOR is typical of Boolean functions: "almost all" Boolean functions have message complexity $\Theta(n \log n)$.

THEOREM 6.7. *Let $n = 2^{2k}$. The probability that a random computable Boolean function on n variables can be computed on a ring of size n in less than $(n/64)ln(n/64)$ messages is bounded by*

$$2^{1-2^{\sqrt{n}}/n}.$$

PROOF. Let h the homomorphism defined by

$$h(0) \rightarrow 01, \qquad h(1) \rightarrow 10.$$

Each string of length 2 occurs in $h^3(0)$ and $h^3(1)$. Let σ_1 and σ_2 be strings of length $2^k = \sqrt{n}$, and let $\omega_i = h^k(\sigma_i)$, $i = 1, 2$. According to Corollary 6.6, if $f(\omega_1) \neq f(\omega_2)$, then at least

$$\frac{n}{32} \ln \frac{n}{8\sqrt{n}} = \frac{n}{64} \ln \frac{n}{64}$$

messages are needed to compute f. There are $2^{\sqrt{n}}$ distinct strings of length $\sqrt{n}$; hence $2^{\sqrt{n}}$ distinct strings of length n are obtained by k applications of h. A computable Boolean function can be computed in fewer messages than the above bound only if it obtains the same value on all these strings. Using the same argument as in Theorem 5.4 we find that this occurs with probability at most $2^{1-2^{\sqrt{n}}/n}$. □

7. *Lower Bounds for Arbitrary Ring Size*

The lower bounds of the previous section are valid only for a sparse set of values n. There is no simple way of extending these results to arbitrary n: Adding further processors to a "good" configuration may destroy symmetry and thereby invalidate the lower bound argument. In this section we give two methods for building strings of arbitrary length with many repetitions of substrings. The first method is suitable for the XOR problem; the second, which is an extension of ideas from [7], is suitable for the orientation and start synchronization problems.

7.1 NONUNIFORM HOMOMORPHISMS. We use the following notation: Let $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$. Then $\mathbf{u} > \mathbf{v}$ if strict inequality holds in each coefficient, that is, $u_i > v_i$ for $i = 1, \ldots, n$; $\mathbf{u} \geq \mathbf{v}$ if $u_i \geq v_i$, for each i. The vector $\mathbf{u}$ is *positive* if $\mathbf{u} > \mathbf{0}$, that is, $u_i > 0$ for $i = 1, \ldots, n$; it is *nonnegative* if $\mathbf{u} \geq \mathbf{0}$. Similarly, a matrix is *positive(nonnegative)* if all its coefficients are positive (nonnegative). The *size* of a positive vector $\mathbf{u}$ is defined to be the sum of its coefficients; this is equal to its l_1 norm $|\mathbf{u}| = \sum_i |u_i|$.

We associate with the string ω its *characteristic vector*

$$\chi_\omega = \begin{pmatrix} a \\ b \end{pmatrix},$$

where a is the number of zeros in ω, and b is the number of ones. χ_ω is a nonnegative vector, and $|\chi_\omega| = |\omega|$.

The homomorphism h is associated with the *characteristic matrix*

$$\mathbf{A}_h = (\chi_{h(0)} \quad \chi_{h(1)}).$$

We have the following basic relation: If $\sigma = h(\omega)$, then

$$\chi_\sigma = \mathbf{A}_h \cdot \chi_\omega.$$

Conversely, if $\mathbf{u} = \mathbf{A}\mathbf{v}$, where $\mathbf{u}$ and $\mathbf{v}$ are nonnegative integer vectors and $\mathbf{A}$ is a nonnegative integer matrix, then we can define strings ω and σ and a homomorphism h such that $\mathbf{A} = \mathbf{A}_h$, $\mathbf{u} = \chi_\sigma$, $\mathbf{v} = \chi_\omega$, and $\sigma = h(\omega)$. Thus we can relate the behavior of iterated string homomorphisms to the behavior of iterated linear mappings.

We intend to build a string of arbitrary size n that is obtained by $\Omega(\log n)$ iterated applications of a homomorphism, that is, build a nonnegative integer vector $\mathbf{v}$ of size n such that $\mathbf{v} = \mathbf{A}^k\mathbf{u}$, where $k = \Omega(\log n)$ and $\mathbf{u}$ is integer and nonnegative. We do this in the reverse: We find conditions on $\mathbf{A}$ so that $\mathbf{A}^{-1}$ maps integer vectors into integer vectors; and we find conditions on $\mathbf{A}$ and $\mathbf{u}$, so that $\mathbf{A}^{-k}\mathbf{u}$ is positive after "many" ($\Omega(\log |\mathbf{u}|)$) applications of $\mathbf{A}^{-1}$.

These goals cannot be achieved with the matrix of a uniform homomorphism. It turns out, however, that nonuniform homomorphisms that correspond to positive nonsingular matrices are *quasi-uniform*; a homomorphism h is quasi-uniform if there exists positive constants μ, c_1, and c_2 such that

$$c_1\mu^k \le |h^k(\epsilon)| \le c_2\mu^k, \tag{7a}$$

for any k, and $\epsilon = 0, 1$. This is sufficient to prove that substrings of length k occur $\Omega(n/k)$ times, if they occur at all.

LEMMA 7.1. *Let A be a nonsingular matrix with integer positive coefficients. Then*

(*i*) *The matrix A has two real eigenvalues μ and ν such that $\mu > 1$, and $\mu > |\nu|$.*
(*ii*) *If $\mathbf{u}$ is an eigenvector with eigenvalue μ, then either $\mathbf{u}$ or $-\mathbf{u}$ are positive.*
(*iii*) *There exist positive constants c_1, c_2 such that*

$$c_1\mu^k|\mathbf{v}| \le |\mathbf{A}^k\mathbf{v}| \le c_2\mu^k\,|\mathbf{v}|$$

for any positive vector $\mathbf{v}$.

PROOF. Let

$$A = \begin{pmatrix} a & c \\ b & d \end{pmatrix}.$$

The values of μ and ν can be found by computing the characteristic roots of A. We have

$$\mu, \nu = \frac{a + d \pm \sqrt{(a-d)^2 + 4bc}}{2}, \tag{7b}$$

which implies (i).

If $\binom{r}{s}$ is an eigenvector of A with eigenvalue μ, then

$$(a-\mu)r + cs = 0 \qquad \text{and} \qquad br + (d-\mu)s = 0.$$

We get from eq. (7b) that $\mu > a$ and $\mu > d$. Thus $r > 0$ iff $s > 0$, which implies (ii).

Let $\mathbf{u} = \binom{u_1}{u_2}$ be a positive eigenvector with eigenvalue μ. Let $\mathbf{v} = \binom{v_1}{v_2}$ be an arbitrary positive vector, and let $|\mathbf{v}| = v_1 + v_2 = n$. Then

$$\mathbf{v} \le \frac{n}{\min(u_1, u_2)}\,\mathbf{u}.$$

As $\mathbf{A}$ is positive, it follows that

$$0 < \mathbf{A}^k\mathbf{v} \le \frac{n}{\min(u_1, u_2)}\,\mathbf{A}^k\mathbf{u} = \frac{n}{\min(u_1, u_2)}\,\mu^k\mathbf{u}$$

so that

$$|\mathbf{A}^k\mathbf{v}| \le \frac{|\mathbf{u}|}{\min(u_1, u_2)} \mu^k n.$$

Conversely,

$$\mathbf{A}\mathbf{v} = \begin{pmatrix} av_1 + cv_2 \\ bv_1 + dv_2 \end{pmatrix} \ge \begin{pmatrix} n \\ n \end{pmatrix} \ge \frac{n}{\max(u_1, u_2)} \mathbf{u}.$$

It follows that

$$A^k\mathbf{v} \ge \frac{n}{\max(u_1, u_2)} A^{k-1}\mathbf{u} = \frac{n}{\max(u_1, u_2)} \mu^{k-1}\mathbf{u},$$

and

$$|A^k\mathbf{v}| \ge \frac{|\mathbf{u}|}{\max(u_1, u_2)} \mu^{k-1} n.$$

This proves (iii). □

We assume henceforth that h is a homomorphism that fulfills conditions (7a) and (6c); that is, h is quasi-uniform and any string of length 2 occurs in $h^c(0)$ and in $h^c(1)$, for some c. The theorems that were proved for uniform homomorphisms are essentially valid when uniformity is replaced by quasi-uniformity. We state the claims and leave the simple modifications in the corresponding proofs to the reader.

LEMMA 7.2. *Any string π of length 2 occurs at least $c_1\mu^{k-c}$ times in $h^k(0)$ and in $h^k(1)$, for $k \ge c$.*

LEMMA 7.3. *If σ is a string of length $|\sigma| \le c_1\mu^k$ that occurs cyclically in $h^k(\rho)$, then σ occurs in $h^i(\pi)$, where $|\pi| \le 2$ and $i = \lceil \log_\mu(|\sigma|/c_1)\rceil$.*

THEOREM 7.4. *Let $a = c_1/(c_2\mu^c)$ and $b = c_1^2/(c_2\mu^{c+1})$. Then the following holds. If σ is a string of length $|\sigma| \le a\,|\omega|/|\rho|$ that occurs cyclically in $\omega = h^k(\rho)$, then σ occurs at least $b\,|\omega'|/|\sigma|$ times in any string $\omega' = h^k(\rho')$.*

Let $\mathbf{A}$ be a positive, nonsingular 2×2 matrix. Let $\mathbf{u}$ be a positive eigenvector of $\mathbf{A}$ with eigenvalue $\mu > 1$ such that $|\mathbf{u}| = n$. Then $\mathbf{A}^{-k}\mathbf{u}$ is positive too for arbitrary k. Assume $|\det(\mathbf{A})| = 1$. Then $\mathbf{A}^{-1}$ has integer coefficients. If $\mathbf{v}$ has integer coefficients, then $\mathbf{A}^{-1}\mathbf{v}$ has integer coefficients too. However, we cannot have a vector $\mathbf{u}$ that both has integer coefficients and is an eigenvector of $\mathbf{A}$. Nevertheless, by choosing $\mathbf{u}$ to be a vector of size $|\mathbf{u}| = n$, with integer coefficients, that is a close approximation to an eigenvector of $\mathbf{A}$, we can guarantee that $\mathbf{A}^{-k}\mathbf{u}$ is still positive for large enough k. Thus we can build a positive vector of arbitrary size n from a positive vector of size $O(\sqrt{n})$ by repeated applications of $\mathbf{A}$. The validity of this construction is proved in the next theorem.

THEOREM 7.5. *Let $\mathbf{A}$ be a 2×2 matrix with positive integer coefficients such that $|det(\mathbf{A})| = 1$. Let $\mathbf{w}_0$ be a positive eigenvector of $\mathbf{A}$ with eigenvalue $\mu > 1$ such that $|\mathbf{w}_0| = 1$. Then there exist a constant $c > 0$ such that the following holds:*

If $\mathbf{u}$ is a positive vector with $|\mathbf{u}| = n$ and $|\mathbf{u} - n\mathbf{w}_0| \le a$, then there exists a k such that $\mathbf{v} = \mathbf{A}^{-k}\mathbf{u}$ is a positive vector and $|\mathbf{v}| \le c\sqrt{an}$.

PROOF. Let $\Delta = \det(\mathbf{A})$, with $|\Delta| = 1$, and let μ and $\nu = \Delta/\mu$ be the eigenvalues of $\mathbf{A}$, with $\mu > 1$. Let $\mathbf{w}_1$ be a second eigenvector of $\mathbf{A}$, independent of $\mathbf{w}_0$, such that $|\mathbf{w}_1| = 1$. The eigenvalues of $\mathbf{A}^{-1}$ are μ^{-1} and ν^{-1}, and $0 < \mu^{-1} < |\nu^{-1}| = \mu$. $\mathbf{w}_0$ and $\mathbf{w}_1$ are eigenvectors of $\mathbf{A}^{-1}$, with eigenvalues μ^{-1} and ν^{-1}, respectively.

There is a constant c' (which depends only on $\mathbf{w}_0$ and $\mathbf{w}_1$) such that for any vector $\mathbf{w}$, if $\mathbf{w} := r\mathbf{w}_0 + s\mathbf{w}_1$, then $|r| + |s| < c'|\mathbf{w}|$. It follows that

$$|\mathbf{A}^{-k}\mathbf{w}| = |r\mu^{-k}\mathbf{w}_0 + s\nu^{-k}\mathbf{w}_1| \le c'\mu^k|\mathbf{w}|.$$

We have $\mathbf{u} = n\mathbf{w}_0 + \boldsymbol{\delta}$, where $|\boldsymbol{\delta}| \le a$. Thus

$$\mathbf{A}^{-k}\mathbf{u} = n\mathbf{A}^{-k}\mathbf{w}_0 + \mathbf{A}^{-k}\boldsymbol{\delta} = n\mu^{-k}\mathbf{w}_0 + \mathbf{A}^{-k}\boldsymbol{\delta} \ge n\mu^{-k}\mathbf{w}_0 - |\mathbf{A}^{-k}\boldsymbol{\delta}|\begin{pmatrix}1\\1\end{pmatrix}$$
$$\ge n\mu^{-k}\mathbf{w}_0 - c'\mu^k|\boldsymbol{\delta}|\begin{pmatrix}1\\1\end{pmatrix} \ge n\mu^{-k}\mathbf{w}_0 - c'\mu^k a\begin{pmatrix}1\\1\end{pmatrix}.$$

Thus, if $\mathbf{w}_0 = \binom{p}{q}$, then $\mathbf{v} = \mathbf{A}^{-k}\mathbf{u}$ is positive for

$$k = \frac{1}{2}\log_\mu\left(\frac{\min(p, q)n}{c'a}\right).$$

By inequality (7a)

$$n = |\mathbf{u}| \ge c_1\mu^k|\mathbf{v}| \ge \left(\frac{c_1}{\mu}\sqrt{\frac{\min(p, q)}{c'}}\right)\sqrt{\frac{n}{a}}\,|\mathbf{v}|.$$

The claim follows. □

Note that if

$$\mathbf{A} = \begin{pmatrix} a & c \\ b & d \end{pmatrix}$$

is the characteristic matrix of a uniform homomorphism, then $a + b = c + d$, which implies (for positive integer matrices) that $|\det(\mathbf{A})| \ne 1$. Thus, the last theorem does not apply to uniform homomorphisms.

7.1.1 *Computing XOR.* We now apply the last theorems to XOR. Consider the following homomorphism:

$$h(0) \to 011, \qquad h(1) \to 10.$$

Any substring of length 2 occurs in $h^3(0)$ and in $h^3(1)$, and $\det(\mathbf{A}_h) = -1$. Let $\mathbf{w}$ be a positive eigenvector of A_h of weight n, with eigenvalue $\mu > 1$. Let $\mathbf{w}_1 = \binom{p}{q}$ be an integer vector of weight n nearest to $\mathbf{w}$ (in l_1 norm); let $\mathbf{w}_2 = \binom{p-1}{q+1}$. Then $|\mathbf{w} - \mathbf{w}_i| < 3$, for $i = 1, 2$, and both $\mathbf{w}_1$ and $\mathbf{w}_2$ are positive. Let I_1 and I_2 be binary strings with characteristic vectors $\mathbf{w}_1$ and $\mathbf{w}_2$, respectively. I_1 and I_2 are strings of length n, and the number of ones in I_1 differs by 1 from the number of ones in I_2; $\text{XOR}(I_1) \ne \text{XOR}(I_2)$. By Theorem 7.5, I_1 and I_2 can be obtained by repeated applications of h on strings of length $O(\sqrt{n})$. By Theorem 7.4, there exists positive constants a and b such that every string σ of length $\le a\sqrt{n}$ that occurs in I_1 or in I_2 occurs in I_1 and in I_2 at least $bn/|\sigma|$ times. It follows that every algorithm for XOR sends at least

$$\sum_{2k+1\le a\sqrt{n}} \frac{bn}{2k+1} = \Omega(n\log n)$$

messages in the worst case.

7.2 TWO STAGE STRING CONSTRUCTION. In order to prove an $\Omega(n \log n)$ lower bound for orientation and start synchronization, we need to use another method based on ideas of [7]. Strings are built in two stages. We first use a uniform homomorphism h that fulfills conditions (6c) and (6d) to build a string $\omega' = h^k(0)$. k is chosen such that $|\omega'| = \Theta(\sqrt{n})$. Next, each letter ϵ in ω' is replaced by a string $H(\epsilon)$ of length $\Theta(\sqrt{n})$, thus obtaining a string $\omega = H(\omega')$ of length n. Substrings of length r' occur $\Omega(\sqrt{n}/r')$ times in ω'. This implies that substrings of length $r = r'\sqrt{n}$ occur $\Omega(\sqrt{n}/r') = \Omega(n/r)$ times in ω. The precise result is stated in the next lemma.

LEMMA 7.6. *Let h be a uniform homomorphism that fulfills conditions* (6c) *and* (6d). *Let H be an arbitrary homomorphism with*

$$m \leq |H(0)|, |H(1)|.$$

Let σ be a string that occurs cyclically in $\omega = H(h^k(0))$ such that $|\sigma| \leq m(d^{k-c} - 2)$. Then σ occurs in ω at least

$$\frac{d^{k-c-1}m}{|\sigma| + 2m}$$

times.

PROOF. Let σ' be the shortest cyclic substring of $h^k(0)$ such that σ occurs in $H(\sigma')$. Then

$$|\sigma'| \leq \frac{|\sigma|}{m} + 2$$

so that

$$|\sigma'| \leq d^{k-c}.$$

By Theorem 6.3 the number of occurrences of σ' in $h^k(0)$ is at least

$$\frac{d^{k-c-1}}{|\sigma'|} \geq \frac{d^{k-c-1}m}{|\sigma| + 2m}. \qquad \square$$

COROLLARY 7.7. *Let h be a homomorphism that fulfills conditions* (6c) *and* (6d). *Let H be an arbitrary homomorphism with*

$$m \leq H(0), H(1) \leq \lambda m.$$

Let

$$a = \frac{1}{\lambda d^{c+1}}, \qquad b = \frac{1}{3\lambda d^{c+1}}.$$

Then any string σ of length $m \leq |\sigma| \leq a|\omega|$ that occurs cyclically in $\omega = H(h^k(0))$ occurs at least $b|\omega|/|\sigma|$ times in ω.

We need to show that every string of length n can be obtained by a suitable choice of the lengths of $H(0)$ and $H(1)$. This is done with the help of the following lemma.

LEMMA 7.8. *Let p and q be two positive integers such that $(p, q) = 1$. Then for every n there exist integers r, s such that*

$$rp + sq = n \qquad (7c)$$

and

$$|r - s| \leq \frac{p+q}{2}. \tag{7d}$$

PROOF. The existence of numbers r, s that fulfill (7c) follows from elementary number theory. Let r, s fulfill (7c), and assume that $|r - s| > (p + q)/2$. Assume without loss of generality that $r > s$. The pair $r' = r - q$ and $s' = s + p$ also fulfills (7c), and $|r' - s'| < |r - s|$. Thus, if we pick a pair r, s that fulfills (7c) and minimizes the difference $r - s$, the pair satisfies (7d). □

7.2.1 *Orientation.* Let $\omega = \epsilon_1 \cdots \epsilon_n$ be a binary string with an even number of ones. We associate with ω two ring orientations D^a and D^b, defined by $D_i^a = \epsilon_1 \oplus \cdots \oplus \epsilon_i$, and $D_i^b = \overline{D_i^a}$. Both ring orientations fulfill the recurrence relation

$$D_i = D_{i-1} \oplus \epsilon_i, \qquad i = 2, \ldots, n.$$

Since $\epsilon_1 \oplus \cdots \oplus \epsilon_n = 0$, we also have

$$D_1 = D_n \oplus \epsilon_1.$$

Also note that

$$D_{i-1} = D_i \oplus \epsilon_i.$$

Hence the values of D_i and $\epsilon_{i-k+1}, \ldots, \epsilon_{i+k}$ uniquely determine the k-neighborhood of processor i in configuration D. It follows that each substring σ of length $2k$ in ω is associated with two (possibly distinct) k-neighborhoods in D^a and D^b; each cyclic occurrence of σ in ω contributes one occurrence of one these two neighborhoods in D^a and one occurrence of the second one in D^b.

If

$$D_i = \bar{D}_j \quad \text{and} \quad \epsilon_{i+s} = \epsilon_{j+1-s}, \qquad s = -k + 1, \ldots, k,$$

then

$$D_{i+s} = \bar{D}_{j-s}, \qquad s = -k, \ldots, k.$$

This implies that processor i and processor j have reverse orientations but identical k-neighborhoods in a ring with orientation D. Taking $j = i - 1$, we find that processors i and $i - 1$ have distinct orientations and identical k-neighborhoods in D^a and D^b when $\epsilon_i = 1$ and $\epsilon_{i-k}, \ldots, \epsilon_{i+k}$ is a palindrome; a synchronous orientation algorithm yields distinct outputs at i and $i - 1$ when run on this configuration.

Assume that ω has the following properties:

(1) Each string σ of length $|\sigma| \leq an$ that occurs cyclically in ω occurs at least $bn/|\sigma|$ times in ω.
(2) ω contains a palindrome with one at its center of length at least an.

Then $SI(D^a, D^b, k) \geq bn/(2k + 1)$, for $2k + 1 \leq an$. Assume that the palindrome in ω is centered at location i. Then the four $\lfloor (an - 1)/2 \rfloor$-neighborhoods of processors i and $i - 1$ in D^a and D^b are all identical; processors i and $i - 1$ have distinct outputs in the computation on D^a and in the computation on D^b. Hence, there is a processor P_1 in D^a and a processor P_2 in D^b such that P_1 and P_2 have the same $\lfloor (an - 1)/2 \rfloor$-neighborhoods in their respective configuration, but different outputs. It follows that D^a and D^b are a fooling pair for any orientation algorithm, with $2\alpha + 1 = an$ and $\beta(k) = bn/(2k+1)$; the orientation algorithm uses $\Omega(n \log n)$ messages on at least one of these two configurations.

We proceed now to build a string ω of arbitrary odd length n with the required properties.

Let h be the homomorphism defined by

$$h(0) \rightarrow 00100, \qquad h(1) \rightarrow 11011.$$

Each string of length 2 occurs in $h^2(0)$ and in $h^2(1)$, so that h fulfills conditions (6c) and (6d); $h(0)$ and $h(1)$ are both palindromes and, by induction, $h^k(0)$ and $h^k(1)$ are palindromes. Let

$$\omega' = h^{2k}(0),$$

where

$$k = \left\lfloor \frac{\log_5 n - 1}{4} \right\rfloor.$$

Let p be the number of zeros and q be the number of ones in ω'. One can show by induction on k that

$$p = \frac{5^{2k} + 3^{2k}}{2}, \qquad q = \frac{5^{2k} - 3^{2k}}{2}.$$

It follows that p is odd, q is even, $(p, q) = 1$, and $\sqrt{n}/112 < q < p < \sqrt{n}/4$. By Lemma 7.8 we have integers r, s such that

$$rp + sq = n, \qquad \text{and} \qquad |r - s| \le \frac{p+q}{2} = \frac{5^{2k}}{2} < \frac{\sqrt{n}}{4}.$$

It follows that

$$\max(r, s) \le \frac{n}{2q} + \frac{p+q}{2} < 56.25\sqrt{n},$$

and

$$\min(r, s) \ge \frac{n}{2p} - \frac{p+q}{2} > 1.75\sqrt{n}.$$

If s is even, we replace s by $s + p$, which is odd, and r by $r - q$. We still have $rp + sq = n$ and $56.5\sqrt{n} > r, s > 1.5\sqrt{n}$. Define

$$H(0) = 0^r \qquad \text{and} \qquad H(1) = 1^s,$$

and let

$$\omega = H(\omega') = H(h^{2k}(0)).$$

Then ω has length n. By Corollary 7.7 any string σ of length $a_1\sqrt{n} \le |\sigma| \le a_2 n$ that occurs in ω occurs at least $b|\omega|/|\sigma|$ times, for some positive constants a_1, a_2, and b. We also have

$$\omega = H(h^{2k-1}(0))H(h^{2k-1}(0))H(h^{2k-1}(1))H(h^{2k-1}(0))H(h^{2k-1}(0)).$$

The string $H(h^{2k-1}(0))$ is a palindrome of length $> n/6$ with 1 at its center. Hence ω has all the required properties.

7.2.2 *Start Synchronization.* We prove an $\Omega(n \log n)$ lower bound for start synchronization on rings of size $n = 2m$, for arbitrary m. We use the same homomorphism h as in Section 6.3:

$$h(0) \rightarrow 011, \qquad h(1) \rightarrow 100.$$

Let

$$\kappa = \left\lfloor \frac{\log_3 m - 1}{4} \right\rfloor, \qquad \omega' = h^{2k}(0), \qquad \text{and} \qquad \chi_{\omega'} = \begin{pmatrix} p \\ q \end{pmatrix}.$$

Then $p = \lceil 3^{2k}/2 \rceil$, $q = \lfloor 3^{2k}/2 \rfloor$, and $|p - q| = 1$. We can, by Lemma 7.8, pick r_0 and s_0 so that $pr_0 + qs_0 = m = n/2$ and $|r_0 - s_0| \leq (p + q)/2 = 3^{2k}/2$. Let $r_1 = r_0 + q$ and $s_1 = s_0 - p$. Then $pr_1 + qs_1 = m = n/2$, $\sqrt{n}/45 \leq p, q \leq \sqrt{n}/4$, and $\sqrt{n} \leq r_0, r_1, s_0, s_1 \leq 23\sqrt{n}$.

Let $H(0) = 0^{r_0}1^{r_1}$ and $H(1) = 0^{s_0}1^{s_1}$. Let $\omega = H(\omega')$. The adversary associates with $\omega = \epsilon_1, \ldots, \epsilon_n$ a starting configuration I_ω, as in Section 6.3: Processor i is awakened one cycle later than processor $i - 1$ if $\epsilon_i = 1$ and one cycle earlier if $\epsilon_i = 0$. We have

$$\chi_\omega = \begin{pmatrix} r_0 & s_0 \\ r_1 & s_1 \end{pmatrix} \begin{pmatrix} p \\ q \end{pmatrix}.$$

Thus ω contains m zeros and m ones, so that the starting configuration I_ω is legal.

ω can be rewritten as

$$H(h^{2k-1}(0))H(h^{2k-1}(1))H(h^{2k-1}(1)).$$

It is easy to check that the number of zeros in $H(h^{2k-1}(1))$ is not equal to the number of ones in this string. The adversary forces any correct algorithm for start synchronization to perform at least $|H(h^{2k-1}(1))|/2 = \Omega(n)$ active cycles on I_ω. From Corollary 7.7 it follows that there exist positive constants a_1, a_2 and b such that any string σ of length

$$a_1\sqrt{n} \leq |\sigma| \leq a_2 n$$

occurs in ω at least $bn/|\sigma|$ times. The last two claims imply an $\Omega(n \log n)$ lower bound on the number of messages sent.

8. *Discussion and Conclusion*

The results of this paper, together with the new results of Moran and Warmuth [10] and of Attiya and Mansour [2], characterize the possible complexity functions associated with distributed computations in a ring. "Most" Boolean functions have message and bit complexity $\Theta(n^2)$ in the asynchronous model. Any nonconstant function has bit complexity $\Omega(n \log n)$ [10]. For any "nice" function f in the range $\log n \leq f(n) \leq n$ one can define a family of Boolean functions with bit complexity $\Theta(nf(n))$.

In the synchronous model "most" problems have message and bit complexity $\Omega(n \log n)$; it is possible to exhibit problems with complexity functions anywhere in the range n to $n \log n$ [2].

The input distribution algorithm we gave for the synchronous model takes $\Theta(n \log n)$ bit messages and exponential time. If run synchronously, the input collection algorithm that was given for the asynchronous model uses $\Theta(n^2)$ bit messages and linear time. This exhibits a trade-off between bits and time in the synchronous model. The first algorithm uses a minimum number of bit messages, whereas the second uses a minimum amount of time. It is easy to show, by comparing the number of distinct configurations with the number of distinct computations, that the time t and the number of bit messages m used by an input distribution algorithm are related by $t \geq (m/n)2^{cn^2/m}$, for some positive constant c. The synchronous input collection algorithm given in this paper does not achieve this time bound.

The $\Omega(n \log n)$ lower bounds on synchronous computations for XOR, orientation, and start synchronization can be extended to labeled rings, where each processor has a distinct label, under the same conditions as in [7]: the domain of labels has to be large enough. Using Ramsey's theorem, one reduces general distributed algorithms to algorithms that depend only on input values, the relative orientations, and the relative order of the labels; one next defines "label-producing homomorphisms" that produce strings with symmetric configurations, in terms of both the input values and the relative order of labels. The full details are given in [3].

The lower bounds in the synchronous and asynchronous model are valid even if not all processors are required to have an output. The lower bounds for AND in the asynchronous model and XOR in the synchronous model are valid even if it is only required that a nonempty set of processors output the correct answer, whereas the remaining processors output a special "don't know" symbol. The lower bounds for orientation is valid even if it is only required that a nonempty set of processors achieve a consistent orientation, whereas the remaining processors output a "don't know" symbol. The lower bound for start synchronization is valid even if it is only required that a nonempty subset of the processors output a one simultaneously, whereas the remaining processors output zero, not necessarily simultaneously.

The construction of strings with many symmetries used tools from the theory of DOL languages [13]. The homomorphism applied in the lower bound proofs had to be *repetitive* in the sense that, if a subword σ occurs in a word ω defined by iterating the homomorphism, then σ has to occur $\Omega(|\omega|/|\sigma|)$ times in that word. The conditions we placed on h ensured that h would be repetitive, but these conditions can be weakened and one might be able to give tight conditions for repetitiveness of a homomorphism. Note that, since every subword of ω occurs with high frequency, there can be at most $O(k)$ different subwords of length k in ω, for any length k. Thus, our notion of repetitiveness is related to the notion of *subword complexity* of a language as used in [6], which is the number of different subwords of length k occurring in the words of the language.

We presented two methods for building strings of arbitrary length n: The first uses a nonuniform homomorphism; it generates strings that are repetitive "in the small": Substrings σ of length $|\sigma| \leq \sqrt{n}$ occur $\Omega(n/|\sigma|)$ times. The second method uses two homomorphisms; it generates strings that are repetitive "in the large": Substrings σ of length $|\sigma| \geq \sqrt{n}$ occur $\Omega(n/|\sigma|)$ times. The second method is a generalization of the construction given in [7]. It is easier to use when one needs only one string of a given length; we use it for the orientation and start synchronization problems. When one needs to build several similar repetitive strings, the second method cannot be used. Thus, the first method is needed for XOR. Both methods illustrate the power of the use of iterated homomorphisms to create strings with many repeated substrings.

ACKNOWLEDGMENTS. Our work benefited from discussions with many persons, in particular, Gerhard Buntrock, Danny Dolev, Nati Linial, Shomo Moran, Ira Pohl, and Eli Shamir.

REFERENCES

1. ANGLUIN, D. Local and global properties in networks of processors. In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing* (Los Angeles, Calif., Apr. 28–30). ACM, New York, 1980, pp. 82–93.
2. ATTIYA, H., AND MANSOUR, Y. Language complexity on the synchronous anonymous ring. *Theoret. Comput. Sci. 53* (1987), 169–185.
3. ATTIYA, H., SNIR, M., AND WARMUTH, M. Computing on an anonymous ring. Tech. Rep. UCSC-CRL-85-3. Computer Research Laboratory, Univ. of California, Santa Cruz, Santa Cruz, Calif., Nov. 1985.
4. BURNS, J. E. A formal model for message passing systems. Tech. Rep. 91, Computer Science Dept., Indiana Univ., Bloomington, Ind., 1980.
5. DOLEV, D., KLAWE, M., AND RODEH, M. An $O(n \log n)$ unidirectional distributed algorithm for extrema-finding in a circle. *J. Algorithms 3*, 3 (Sept. 1982), 245–260.
6. EHRENFEUCHT, A., LEE, K. P., AND ROZENBERG, G. Subword complexity of various classes of deterministic developmental languages without interactions. *Theoret. Comput. Sci. 1* (1975), 59–75.
7. FREDERICKSON, G. N., AND LYNCH, N. A. Electing a leader in a synchronous ring. *J. ACM 34*, 1 (Jan. 1987), 98–115.
8. HIRSHBERG, D. S., AND SINCLAIR, J. B. Decentralized extrema-finding in circular configurations of processors. *Commun. ACM 23*, 11 (Nov. 1980), 627–628.
9. ITAI, A. The circular extrema problem with nondistinct numbers. Unpublished manuscript.
10. MORAN, S., AND WARMUTH, M. Gap theorems for distributed computations. Tech. Rep. UCSC-CRL-86-1, Computer Research Laboratory, Univ. of California, Santa Cruz, Santa Cruz, Calif., Jan. 1986.
11. PACHL, J., KORACH, E., AND ROTEM, D. Lower bounds for distributed maximum-finding algorithms. *J. ACM 31*, 4 (Oct. 1984), 905–918.
12. PETERSON, G. L. An $O(n \lg n)$ unidirectional algorithm for the circular extrema problem. *Trans. Program. Lang. Syst. 4*, 4 (1982), 758–762.
13. ROZENBERG, G., AND SALOMAA, A. *The Mathematical Theory of L Systems.* Academic Press, Orlando, Fla., 1980.
14. THUE, A. Über Unendliche Zeichenreihen. In *Videnskapsselskapets Skrifter. I. Mat.-naturv. Klasse.* Kristiania, Norway, 1906, pp. 1–20.
15. THUE, A. Über die Gegenseitige Lage Gleicher Teile Gewisser Zeichenreihen. *Videnskapsselskapets Skrifter. I. Mat.-naturv. Klasse.* Kristiania, Norway, 1912, pp. 1–67.

RECEIVED AUGUST 1985; REVISED FEBRUARY 1987 AND JANUARY 1988; ACCEPTED JANUARY 1988