Distributed Computing on Transitive Networks: The Torus*

Paul W. Beame[†] Hans L. Bodlaender[‡]

July 1988

Abstract

We identify the class of transitive networks as being of particular interest for distributed computation with known topology. This class includes the ring and the complete network as well as the 2-dimensional grid network with boundary connections, i.e. the torus. We consider the bit and message complexity of computing non-constant functions on an asynchronous torus where processors are identical except that they have one input bit. We show that every function can be computed with bit complexity $O(n\sqrt{n})$ on an $\sqrt{n} \times \sqrt{n}$ torus and show that this bound is optimal for many functions including AND, OR and XOR. We also give non-constant functions with a bit complexity linear in the number of processors on the torus. These bounds are better than those possible for the ring network.

1 Introduction

Suppose one were designing a network for distributed computation. One would like to minimize the communication hardware so keeping the degree of the network small would be a high priority. Next, it would be a very natural feature to have each processor in such a network be as much like every other processor as possible. We capture this notion by defining a class of networks, the transitive networks, which have the property that each processor's 'view' of the network is exactly like that of any other processor. A specific low-degree transitive network, the ring network, has been the subject of a great deal of study but we show that the torus, another low-degree transitive network, has better complexity characteristics.

In particular, we consider the torus consisting of n processors arranged in a $\sqrt{n} \times \sqrt{n}$ grid with connections on the boundary. Processors do not know their relative addresses in the torus and there is no processor designated as leader but the size of the torus is known to them. We assume further that the network is asynchronous and that initially processors have exactly one input bit but are further identical. We study the bit and

*A large part of this research was done while both authors were at the Laboratory for Computer Science of the Massachusetts Institute of Technology.

[†]Author's present address: Computer Science Department, University of Washington, Sieg Hall, Seattle, Washington 98195, USA. Research supported in part by NSF grant PYI-25800.

[‡]Author's present address: Department of Computer Science, University of Utrecht, P.O. Box 80.089, 3508 TB Utrecht, the Netherlands. Research supported in part by the Netherlands Organization for Scientific Research N.W.O.

message complexity of computing non-constant functions on this network. These problems have been studied to a large extent for the ring network (see e.g. [1,2,4,5]). As the torus can be seen as a 2-dimensional version of the ring, it is interesting to compare the results.

For every n, all non-constant functions on a ring with size n have bit-complexity $\Omega(n \log n)$ [5] even if processors also have unique identities taken from a relatively small set [3]. For infinitely many n, all non-constant functions on a ring with size n have message complexity $\Omega(n \log^\star n)$ [4]. There are functions (including AND, OR and XOR) with bit and message complexity $\Omega(n^2)$ [2]. Each of these bounds is sharp [2,5].

In contrast, for every n, there exist non-constant functions on the $\sqrt{n} \times \sqrt{n}$ torus that have bit and message complexity $O(n)$. Thus, where there is a "gap" between the size of the ring and the bit and message complexity of computing non-constant functions on it [5], there is no such gap for the torus.

Also, we give a non-trivial algorithm to compute every non-constant function with bit complexity $O(n\sqrt{n})$. We show that AND, OR, XOR and almost any random function has message complexity $\Omega(n\sqrt{n})$. Thus, our bound is sharp.

2 Transitive Networks and the Torus

Definition 2.1 *A* network *consists of an undirected graph $G(V, E)$ with a processor at each node and an assignment of labels from a label set $\mathcal{L}$ to each endpoint of each edge in E which assigns distinct labels to the endpoints surrounding a single node. A processor communicates with its neighbors by sending or receiving messages along incident edges in the network, specifying edges according to the labels on their endpoints at its node.*

The networks of interest are those with *known topology*, i.e. networks in which each processor has encoded in its program a description of the underlying network. However, processors will not know their absolute position within this network except by communicating with their neighbors. The communication along the edges may in general be *asynchronous* but there is order maintained in that edges act as FIFO queues in each direction. An *input configuration* for a network consists of an assignment $i : V \to \{0, 1\}$ of binary input values, one input per processor. Unless otherwise specified our networks are *anonymous*, those for which the programs of the nodes are all identical. (A network has *identities* if the programs are identical except for distinct values chosen from a set $\mathcal{I}$ of possible values.) We will make no other restrictions on the programs of the processors.

Definition 2.2 *An* automorphism *of a network is a mapping of the node-set V of the network to itself which preserves both incidence and labeling.*

A function f on the network maps the input configuration i to a single bit. Since processors have no inherent notion of an absolute location in the network, this function must be invariant under automorphisms of the network, i.e. if ϕ is an automorphism of the network then $f(i) = f(i \circ \phi)$. A network computes f if, when started with input configuration i, the processors eventually halt with $f(i)$ stored in each processor (whenever f is defined.) The *message complexity* for computing f is the worst-case number of messages exchanged in order to compute f on any input configuration. The *bit complexity* for computing f counts the sum of the lengths of these messages in bits rather than simply their number.

Notice that if the network has some processor whose neighborhood looks different from that of the other processors, this can be exploited to select that processor as a leader who can collect the inputs, compute f, and distribute its value to the other processors. In such a case, the algorithm loses much of its distributed flavor so it is more interesting when this is not possible.

Definition 2.3 *A network is* transitive *if, for each pair of nodes u and v, there is an automorphism of the network that maps u to v.*

Transitivity captures the notion that the neighborhood of every node in the graph looks like the neighborhood of any other. Specific networks such as the ring and the complete network are special cases of transitive networks. We concentrate here on a particular family of networks which is the natural transitive analog of the grid networks, namely the *torus* networks.

Throughout this paper whenever we use m we shall assume that it is $\sqrt{n}$ and an integer.

Definition 2.4 *The $m \times m$ torus network consists of m^2 processors indexed by the set $V_m = \{(i,j) \mid 0 \leq i,j \leq m-1\}$ and undirected edges from each processor (i,j) to processors indexed $(i, j \pm 1)$ and $(i \pm 1, j)$ where addition is taken modulo m (giving wrap-around connections). For processor (i,j), its endpoints of its edges to $(i, j+1)$, $(i, j-1)$, $(i+1, j)$, and $(i-1, j)$ are labeled up, down, $left$, and $right$ respectively. A more general $k \times l$ torus network is defined analogously.*

The ring can be viewed as an $n \times 1$ torus but we will see that, if the smaller dimension of the torus is not too small, the torus is a more efficient network than the ring with respect to both bit and message complexity.

3 Any function has bit complexity $O(n\sqrt{n})$ on the torus

In this section we show that any function in our model can be computed with bit-complexity $O(n\sqrt{n})$. We will also show an $\Omega(n\sqrt{n})$ lower bound for the bit-complexity of a number of functions, including AND, OR and XOR. In addition we consider the complexity for non-square tori.

We first give the $O(n\sqrt{n})$ bit-complexity algorithm for computing any function on $\sqrt{n}$ by $\sqrt{n}$ tori. If it were possible to designate one of the processors in the torus as a leader then we could have that processor collect all the input bits, compute the function value, and distribute this value to all the other processors. Unfortunately, since the processors do not have identities, finding a leader will not be always be possible. However, we can make use of the natural asymmetries in the input bits for the problem to select certain of the processors as leader processors. Where there is symmetry in the input bit pattern this can fail, but that symmetry will mean that fewer bits will be needed to specify the entire input. Our algorithm will in fact look for a sub-grid of processors, all of whose neighborhoods look alike, i.e. have the same pattern of input bits. Each processor will then only have to collect the bits from its neighborhood to compute the function and will only have to distribute its answer to its neighborhood. Thus, once this subgrid is found, the rest of the computation can be done with a linear number of bits communicated.

The algorithm consists of a number of distinct phases. As a general rule, each message is labeled with its phase number. A processor that receives a message from a later phase than its current phase, stores the message in a queue for the later phase, and processes the message when it reaches that phase. If a processor receives a message from an earlier phase than its current phase then it forwards it with the appropriate actions it would have taken in that phase.

We will only give an informal description of the algorithm and will describe and analyze the algorithm phase by phase. Note that, despite not being able to tag a message by an identity, a processor can easily stop a phase of a protocol which involves a known number of processors sending messages once around the toroidal connections simply by counting the number of messages that it has received.

Phase 1: Symmetry is broken as much as possible in each row so that certain processors are chosen to be leaders in each row. At the end of this phase the bit strings between each adjacent pair of leaders on the same row will be the same and each processor in the row will have this bit string stored in a local variable *row-pattern*.

Each processor starts the phase by sending its own input bit to its right neighbor. It continues by receiving bits from its left neighbor, recording them, and forwarding them to its right neighbor until m bits (including its own) have been recorded. At this point it has a string consisting of all input bits of the processors on its row. Each such string is a cyclic shift of all the other strings in the same row because the links are FIFO. Now each processor computes the lexically maximal cyclic shift of its string and declares itself to be a row leader if its bit could be at the head of this maximal string. If there is no symmetry in the string there will be one row leader per row; however, if the row can be expressed as w^k for some string $w \in \{0,1\}^*$ there will be k row leaders in it. Since each processor in the row knows the original string, each processor can determine which processors in the row are leaders and exactly what string of bits occurs between each adjacent pair of leaders in its row. Each processor stores this string in variable *row-pattern*.

Clearly, phase one uses mn messages with $O(1)$ bits each.

Phase 2: Let $\prec$ be the following total ordering of bit strings: $s \prec t \;\Leftrightarrow |s| < |t|$ or ($|s| = |t|$ and s is lexically smaller than t). At the end of this phase every processor will know whether or not its *row-pattern* is maximal with regard to $\prec$.

The idea is that each row leader will send its *row-pattern* along with a counter initialized to 1 down its column, each processor that receives this message will compare this pattern with its own – tagging the message with 'kill' if it finds that its own pattern is larger. If this message is the first message that it has seen in this phase then it increments the message's counter. The processor then forwards the message to the processor below it along with the kill tag if necessary. Row leaders keep a running total of the counters of the messages received in this phase.

A message has eventually made it all the way around back to its initiating row leader if and only if that row leader's running total is m. If the *row-pattern* for that row is maximal then the message arrives back without a kill tag and the row leader starts a MAXIMAL message around its row to the right and starts phase 3. If a *row-pattern* is not maximal then a row leader in that row will receive its own pattern back with the kill tag so it will start a SMALL message around its row to the right and starts phase 3. Processors that have not initiated MAXIMAL or SMALL messages simply record and forward them to the right and start phase 3.

To estimate the number of bits sent during phase 2, first note that the total length of all *row-patterns* over all row leaders is n. It follows that the total number of bits, used to send the patterns in the messages is exactly nm. Since each processor in a column increments a message counter exactly once, the values of the counters in a column never add to more than m. Each counter travels only distance m so the total number of bits to communicate all the counters is at most m^2 per column and thus $O(mn)$ in over all. Note that the obvious idea of incrementing each counter and not keeping the running totals could have required $\Omega(nm \log m)$ bits for the counters over all. Finally, as all other information is $O(1)$ bits per message, and there are $\leq O(nm)$ messages sent in this phase, the bit complexity of phase 2 is bounded by $O(nm)$.

Phase 3: In phase 3, a subset of the rows with maximal *row-pattern* is selected, so that all (vertical) distances between a selected row and the next selected one are equal. The selected rows are called *active*, the others *passive*.

The technique is similar to the technique in phase 1, but column-wise, instead of row-wise. A processor sends a 1 downward if it is in a row with maximal *row-pattern* and a 0 downward otherwise. It again forwards the first $m-2$ messages it receives in this phase and stops on the $m-1$-st message, recording the bits as it goes. Note that each processor in a given row will have the same bit string from this phase.

Processors whose strings in this phase are maximal with respect to cyclic shifts select themselves as being on 'active' rows, all others become passive. By choosing to have 1 correspond to maximal *row-pattern*, all active rows will have maximal *row-pattern*. As in phase 1, active rows are equally spaced and processors also can compute the distance between successive active rows, which is stored locally in an integer *dist*.

Clearly phase 3 has bit complexity $O(nm)$.

Phase 4: In this phase some of the active rows become inactive so that between each pair of successive active rows that remain, the sequence of *row-patterns* is the same.

Sending *row-patterns* for passive rows all the way around the torus would take too many bits so we have to find a more efficient method. This will take several subphases. At the end of each subphase active rows will be equally spaced at distance *dist* and for some i, all rows which are distance $j \leq i$ below an active row will have *row-pattern* = *row-pattern*$_j$. Each phase will either at least double *dist* or will increment i and we will stop if either $dist = m$ or $i = dist$. We will 'shade' rows which have their *row-patterns* correctly matched up. Initially each processor in an active row will set a local *shade* bit to 1 and all processors in passive rows will have their shade bits set to 0. A processor will be able to discern which subphase a message belongs to simply by keeping account of the number of messages of each type it has received and holding a message until the appropriate subphase. Each subphase consists of the following 4 steps:

Step 1. The row leaders on active rows check to see if either $dist = m$ (in which case there is only one active row) or all rows have been shaded and begin phase 5 if either condition holds. They can easily check if all rows have been shaded by simply testing if the processor immediately above has its *shade* bit set to 1.

Step 2. Each row leader on an active row now learns the *row-pattern* of the first unshaded row below it, and the *row-pattern* of the first unshaded row below the first active row above it. This *row-pattern* below can be found by sending a message down, which is forwarded until it arrives in a processor with *shade* bit 0, which sends a message back, containing the relevant information. The *row-pattern* above is found by sending a message all the way up to the active row above and then back down through the first unshaded row where the pattern is collected and sent down to the initiating processor.

Step 3. Each row leader on an active row compares the two *row-patterns* it has just collected and sets a local variable *comp* to $\prec$, $\succ$, or $=$ depending on the outcome of this comparison. It then starts a message around its row to the right containing *comp*. Non-leader processors on active rows record the *comp* for their row and forward *comp* to the right (row leaders do not forward *comp* messages since they will be generating their own).

Step 4. We now execute something like phase 3, except only for active rows. That is, each processor on an active row sends *comp* downward and forwards the first $m/dist - 2$ such messages to collect a string of the results of the $m/dist$ comparisons. Processors on passive rows merely forward such messages. As before the strings are all cyclic shifts of each other.

1. If all the signs are equal then they must be $=$ signs (since $\prec\prec\cdots\prec$ and $\succ\succ\cdots\succ$ cannot be obtained) and the processors on the active rows send messages down to turn the *shade* bits of the first unshaded row to 1.

2. If not all signs are equal, using a fixed ordering on $\prec$, $=$, and $\succ$, a processor in an active row can select its row to become passive if its comparison string is not lexically maximal among all possible cyclic shifts. Note that again all active rows are equidistant and each processor in an active row can compute this new distance which is at least twice the old one and store it in *dist*.

Note that in both cases, the invariant stays true.

Since the *row-patterns* of active rows are maximal, in step 2 of one subphase each active row only collects *row-pattern*s whose total length is at most $2m$ and in total such patterns travel distance m. The bit complexity of one subphase is dominated by the cost of step 4 which is bounded by $O(nr)$ for each subphase in which there are r active rows.

Thus, the total work of the subphases in which case 1 of step 4 occurs is bounded by $O(mn)$. (Each active row corresponds to an unshaded row that is shaded. In total we can shade $\leq m$ rows.) In case 2 of step 4 at most half of the active rows stay active so the total work of the subphases in which this case occurs is bounded by $O(mn+\frac{1}{2}mn+\frac{1}{4}mn+\cdots) = O(mn)$.

Phase 5 After the completion of phase 4 the active rows are in the same position with respect to the *row-pattern*s that the row leaders are with respect to the bits in each row. However, the row leaders within a single active row do not share the same view of the rows between them and the next active row below. This is because a row leader's horizontal distance to the beginning of the *row-pattern*s in the rows below may be different from its neighbor's.

In phase 5 we reduce the number of row leaders per active row (i.e. some row leaders cease to be row leaders) so that the length of any *row-pattern* divides the distance between two successive row leaders on an active row.

Each row leader in an active row first learns its horizontal distances to the left of all row leaders in the rows between it and the next active row below, and the length of the *row-pattern*-strings of each of these rows. This can be done easily with $O(nm)$ bits transmitted.

Now each row leader has a string with *dist*−1 numbers containing these horizontal distances. It is easy to compute these strings for all other row leaders in the same row with the help of the lengths of the *row-pattern*s collected. A row leader stays a row leader if and only if its string is lexically maximal with regard to the strings of all other row leaders in its row.

It easily follows that the length of any *row-pattern* divides the horizontal distance between two successive row leaders in an active row and that the distances between all pairs of successive row leaders in active rows are equal. (This is essentially a distributed least common multiple algorithm.) Note that it is important that this occurs after phase 4 because if we had done this earlier we would not necessarily have considered all *row-patterns*.

Phase 6: In this phase the final grid of processors is computed.

After phase 5 each row leader in a given active row has the same view of the rows between it and the next active row. However, these views may differ from active row to active row because the horizontal shifts of the *row-pattern*s on the active rows may be different since the previous phase only normalized shifts locally. We cannot send the collection of horizontal distances around the torus because that would require too many bits. Instead we follow a similar procedure to that in phase 4.

We start the procedure by setting the *shade* bits of all rows (even active ones) to 0 because even the active rows may be irregularly shifted with respect to each other. Define the 'offset' of a row to another row to be the horizontal minimum distance from left to right from a row leader on the first row to a row leader in a second row. Say that two rows are comparable if they are the same distance below an active row. We already know that comparable rows have the same *row-pattern* so that all horizontal left to right distances from the one to the other are equal to the offset. The key difference from phase 4 is that the comparison which the row leaders on active rows make is not between the *row-pattern*s on their surrounding comparable unshaded rows but between the offsets that each of these comparable unshaded rows has from its successor. Each row leader on an active row can calculate this by collecting the horizontal left to right distances along its column to the first unshaded row at or below it, the first such row at or below the active row above it, and the first such row at or below the first active row below it. This computation replaces step 2 of each subphase in phase 4 but otherwise that procedure remains unchanged. Clearly the horizontal distances require no more bits than the *row-pattern*s did in phase 4 so the total bit complexity is at most $O(nm)$.

Phase 7: At the end of phase 6, the patterns of input bits in the region bounded by the columns of two consecutive row leaders in one active row and two consecutive active rows are all the same. Since the row leaders in active rows are also symmetrically placed, learning the bits in this region is sufficient to learn the entire input. From previous computations each row leader in an active row already knows the *row-pattern*s and offsets of each of these rows which is sufficient to determine these bits. Hence each row leader on an active row can compute the outcome of the function on the input. Finally, with $O(n)$ bits sent, this outcome can be broadcasted to all other processors.

Thus we obtain the following theorem:

Theorem 3.1 *Every one-bit-input function on the $\sqrt{n} \times \sqrt{n}$ torus can be computed with* *bit complexity $O(n\sqrt{n})$.*

For the case that the torus is non-square, one can obtain the following result with a very similar algorithm.

Theorem 3.2 *Every one-bit-input function on the $k \times l$ torus can be computed with bit complexity $O(k \cdot l \cdot \max\{k, l\})$.*

Lower bounds

The following lower bound results can be obtained in the same way as similar results for the ring, by Attiya, Snir and Warmuth [2], or by an easy simulation on the ring.

Theorem 3.3 *AND, OR, XOR, SUM have message complexity $\Omega(n\sqrt{n})$ on anonymous $\sqrt{n} \times \sqrt{n}$ tori, and message complexity $\Omega(k \cdot l \cdot \max\{k, l\})$ on anonymous $k \times l$ tori.*

Theorem 3.4 *The probability that a random computable Boolean function on the $\sqrt{n} \times \sqrt{n}$ anonymous torus has message complexity $< \frac{1}{4} n\sqrt{n}$ is less than $2^{1-2^{n/2}/n}$.*

4 A non-constant function on a torus with bit complexity $O(n)$

In this section we show that for all $m = \sqrt{n}$, there exists a non-constant function on an $\sqrt{n} \times \sqrt{n}$ torus, that can be recognized with bit complexity $O(n)$. We also give some results for the case that the torus is non-square.

First, suppose that each processor has one of the following 3 input-values: $\{0, 1, \#\}$.

When the output is '1' we say the processors 'accept' the input pattern, otherwise they 'reject' the input pattern.

Initially, each processor is active. The algorithm consists of phases. In each phase, between $1/4$ and $1/9$ of the active processors stay active, or some processors reject the input. Each message carries the phase number. We apply the same rules as in the previous section for active processors that receive messages of a later phase than their current phase.

If during any phase of the algorithm, a processor decides to reject, then it broadcasts a *reject* messages, and ignores all further messages. Clearly, the total number of *reject* messages is $O(n)$.

At each phase certain processors will check that the processors with the input $\#$ from the previous phase form an appropriately spaced 'subgrid' in the torus. That is, there is a collection of rows R, and a collection of columns C, and a processor has a $\#$, if and only if it is on a row $\in R$, and a column $\in C$. The processors in this subgrid will then each collect two bits locally from processors not in the subgrid to determine the input for the next phase of the algorithm. In each phase of the algorithm, the rule for spacing will be that consecutive rows or columns of the subgrid are either 2 or 3 apart within the previous grid.

In the first phase, each processor has its bit sent to the processors with relative addresses (i, j), with $-3 \leq i, j \leq 0$. If none of the (fifteen) inputs a processor receives is a $\#$, then it rejects. Otherwise, processors with a 0 or 1 input become passive. Processors with a $\#$ input check whether one of the following cases holds:

1. The processors with relative addresses $(0,2)$, $(2,0)$ and $(2,2)$ have input #, and no other processor of which an input is received has input #.
2. The processors with input # are: $(0,2)$, $(3,0)$ and $(3,2)$.
3. The processors with input # are: $(0,3)$, $(2,0)$ and $(2,3)$.
4. The processors with input # are: $(0,3)$, $(3,0)$ and $(3,3)$.

If none of these cases holds, then the processor rejects.

Otherwise, the processor stays active and goes to the next phase. It computes a new input for the following phase as follows. If the processors with relative addresses $(0,1)$ and $(1,0)$ both have input 0 then the new input is 0. If both inputs are 1 then the new input is 1, otherwise it is #.

Note that if no processor rejects in the first phase, then the active processors do indeed form a subgrid in the torus.

In the second (and later) phases, the same procedure can be followed. Now passive processors only forward messages. In each phase, the number of rows and columns containing active processors, drops by a factor between 1/2 and 1/3.

One can show that there exist input patterns such that no processors reject during the first $\lfloor \log m \rfloor$ phases. Necessarily, after $\lfloor \log m \rfloor$ phases, there is at most one active processor. This processor then can start a traversal (e.g. with DFS) to check that no processor rejected. If so, then it accepts the input pattern and starts a broadcast of *accept* messages.

Note that the input of the active processors in phase $i+1$ depends only on the outputs of the active processors in phase i. Thus, no matter how message delays are scheduled, the same output will be generated on a given input pattern. So the algorithm indeed computes a non-constant function.

In each phase, we can partition the torus into rectangles of processors, each with a single active processor at its lower left corner. To estimate the bit complexity of the algorithm note that in phase i only a constant number of messages traverse one of these rectangles in the partition for phase i and that each message has only $O(\log i)$ bits in it. Each message travels a distance within the rectangle proportional to the larger of the two dimensions of the rectangle. By construction the smaller of the two dimensions of the rectangle is at least 2^i so the number of bits sent in each rectangle is an $O(\log i \cdot 2^{-i})$ fraction of the number of processors in it and thus the number of bits sent with phase i messages is

$$O(\log i \cdot \frac{n}{2^i}).$$

Hence, the total bit complexity of the algorithm is bounded by

$$O(n) + O(\sum_{i=1}^{\lfloor \log m \rfloor} \log i \cdot \frac{n}{2^i}) = O(n).$$

In case processors have a one-bit input (and hence input # is impossible), a phase similar to a single phase of the algorithm above can be applied to obtain the initial subgrid of #'s and then the algorithm proceeds as before. In this way the following result is obtained.

Theorem 4.1 *For every $m = \sqrt{n}$, there exists an one-bit-input function on the $\sqrt{n} \times \sqrt{n}$ torus, that can be computed with bit complexity $O(n)$.*

Now suppose we have a torus with k columns and l rows, $k > l$. Using a similar procedure as above, we can obtain, with $O(k \cdot l)$ bits sent, a situation where there is one row that contains $O(k/l)$ active processors. On the other rows there are no active processors. The distances between active processors are $O(l)$. Now one computes a function on the (current) inputs of the active processors, as in [5], with $O(k/l \log(k/l) \cdot l)$ bits sent, or with $O(k/l \log^*(k/l) \cdot l)$ messages sent. Hence, the total bit complexity is $O(k \cdot l + k \log k)$, and the total message complexity is $O(k \cdot l + k \log^* k)$.

Theorem 4.2 *For every k, l, $k > l$, $n = k \cdot l$, there exists an one-bit-input function on the $k \times l$ torus, that can be computed with bit complexity $O(k \cdot l + k \log k) = O(n + k \log k)$ and with message complexity $O(k \cdot l + k \log^* k) = O(n + k \log^* k)$.*

Clearly, this is optimal (and linear) for $l = \Omega(\log n)$ and $l = \Omega(\log^* n)$ respectively. By modifying the lower bound arguments of [5] and [4] for rings we can obtain

Theorem 4.3 *For every $k, l, k > l$, $n = k \cdot l$, every one-bit-input function on the $k \times l$ torus, requires bit complexity $\Omega(k \cdot l + k \log k / \log l) = \Omega(n + n \log n/(l \log l))$ and message complexity $\Omega(k \cdot l + k \log^* k / \log^* l) = \Omega(n + n \log^* n/(l \log^* l))$.*

The lower bounds do not count any bits sent along the smaller dimension of the torus which is certain to account for the $\log l$ and $\log^* l$ factors that separate the upper and lower bounds. Note that the lower bounds show that, for $l = o(\log n / \log \log n)$ and $l = o(\log^* n / \log^* \log^* n)$ respectively, it is impossible to achieve linear upper bounds.

5 Variations of the model

5.1 Processors having identities

For the ring, complex arguments have to be devised to obtain lower bounds for computing non-constant functions where processors have, besides their input bit, a unique identity taken from a reasonably small sized set of possible inputs [3]. For the torus with identities most lower bounds are much easier.

Peterson [6] gave an algorithm to find a leader on a torus with $O(n)$ message complexity. This algorithm relies on the fact that the number of points on the boundary plus the diameter of a 2-dimensional region can grow much more slowly than the number of points in its interior. It follows that every non-constant function can also be calculated with linear message complexity. However, The $O(n\sqrt{n})$ bound of section 3 cannot be improved even with identities, as there are functions that still have $O(n\sqrt{n})$ bit complexity on a torus with processors with identities. (Take for instance the function that outputs a 1, if and only if each row is of the form ww or $ww0$ for some $w \in \{0,1\}^{\lfloor n/2 \rfloor}$. Use an information theoretic argument.)

5.2 Synchronous communication

In case the torus is synchronous, several results for the ring can be carried over without much difficulty. Attiya, Snir and Warmuth [2] showed that XOR and several other functions have a message complexity of $\Omega(n \log n)$ on a synchronous ring. With a simulation argument one can show that there exist functions on the $m \times m$ torus that have a message complexity of $\Omega(m^2 \log m)$. For instance, consider XOR, and suppose m is odd. Give all processors in the same column the same input. All processors in the same column send the same messages. If $o(m \log m)$ messages are sent per row, then by simulation one can obtain an algorithm for computing XOR on the synchronous ring of size m, that sends $o(m \log m)$ messages. Contradiction. Hence $\Omega(m \log m)$ messages are send per row, i.e. $\Omega(m^2 \log m)$ messages in total.

With an algorithm that is very similar to the algorithm in [2] one can show that this bound is sharp, i.e. one can compute every non-constant function on the $m \times m$ torus with message complexity $O(m^2 \log m)$, and by using coding in time, with bit complexity $O(m^2 \log m)$.

5.3 Other networks

The results of section 3 and 4 can be generalized to d-dimensional grids with boundary connections, with $d \geq 3$. One can show that every function on a d-dimensional $m \times \cdots \times m$ grid with boundary connections (with d constant) can be computed with bit complexity $O(m^{d+1}) = O(n^{1+1/d})$. Also, one can devise non-constant functions on these networks with bit complexity linear in the number of processors.

There many potential transitive networks that one could consider. It would be interesting to find such a network, particularly one of low degree, for which one could obtain even smaller bit and message complexity for computing non-constant functions. A particular transitive network that seems to deserve consideration is the hypercube network but the above results do not appear to extend to it.

5.4 Average case analysis

In this paper we did not consider the *average* message and bit complexity of computing functions on the torus. A number of very interesting results on this problem for the ring network were recently obtained by Attiya and Snir [1]. It might be interesting to see whether these results can be extended to the torus. For instance, it might be possible to combine the labeling procedure of Attiya and Snir [1] with Peterson's leader-finding algorithm on the torus [6] in order to find a method with average message complexity $O(m^2)$ to compute any function on an $m \times m$ torus. A similar remark applies for the expected number of messages of randomized algorithms. The average bit complexity of computing non-constant functions apparently is open for the torus, as well as for the ring network.

References

[1] H. Attiya and M. Snir. *Better Computing on the Anonymous Ring.* IBM Tech. Rep., IBM Research Division, Yorktown Heights, NY, 1988.

[2] H. Attiya, M. Snir, and M. Warmuth. Computing on an anonymous ring. In *Proc. 4th Ann ACM Symp. Principles of Distributed Computation*, pages 196–203, 1985.

[3] H. Bodlaender, S. Moran, and M. Warmuth. The inherent complexity of asynchronous computations on non-anonymous rings. 1987. Draft paper.

[4] P. Duris and Z. Galil. Two lower bounds in asynchronous distributed computations. In *Proc. 28th Ann. IEEE Symp. on Foundations of Comp. Science*, pages 326–330, 1987.

[5] S. Moran and M. Warmuth. Gap theorems for distributed computations. In *Proc. 5th Ann ACM Symp. Principles of Distributed Computation*, pages 131–140, 1986.

[6] G. L. Peterson. *Efficient Algorithms for Elections in Meshes and Complete Networks.* Technical Report TR 140, Dept. of Comp. Science, Univ. of Rochester, 1985.