ANONYMOUS WIRELESS RINGS

Krzysztof Diks [*†§]
(diks@uqah.uquebec.ca)

Evangelos Kranakis [‡**]
(kranakis@scs.carleton.ca)

Adam Malinowski [*¶]
(amal@mimuw.edu.pl)

Andrzej Pelc [†||]
(pelc@uqah.uquebec.ca)

Abstract

We introduce anonymous wireless rings: a new computational model for ring networks. In the well-known hardware ring each processor has two buffers, one corresponding to each of its neighbors. In the wireless ring each processor has a single buffer and cannot distinguish which neighbor the arriving bit comes from. This feature substantially increases anonymity of the ring. A priori it is not clear whether any non-trivial computation can be performed on wireless rings. Nevertheless we show that wireless rings are computationally equivalent to hardware rings.

1980 Mathematics Subject Classification: 68Q99
CR Categories: C.2.1
Key Words and Phrases: Anonymous network, Asynchronous algorithm, Input collection, Synchronous algorithm, Wireless ring.

---

[*]Instytut Informatyki, Uniwersytet Warszawski, Banacha 2, 02-097 Warszawa, Poland

[†]Département d'Informatique, Université du Québec à Hull, Hull, Québec J8X 3X7, Canada

[§]Research supported in part by NSERC International Fellowship and by EC Cooperative Action IC 1000 (project ALTEC: Algorithms for Future Technologies).

[‡]Carleton University, School of Computer Science, Ottawa, ON, K1S 5B6, Canada

[**]Research supported in part by NSERC grant 907002.

[||]Research supported in part by NSERC grant OGP 0008136.

[¶]Research supported by EC Cooperative Action IC 1000 (project ALTEC: Algorithms for Future Technologies).

1 Introduction

Due to the simplicity of its topology (small number of links and low branching) the ring has been the focus of investigation in several papers on distributed computing. Issues studied include message and bit complexity of computing boolean functions [3, 4, 10], computations like extrema finding [7, 8], leader election [1, 2, 9, 11], orientation [3], symmetry breaking [12], etc.

The ring model (unidirectional and bidirectional) currently used in the literature is hardware based. Each processor has physical links to both its neighbors. Within this framework there are already several variants of the model and hence of the problems previously mentioned depending on whether or not the ring is oriented, the system is synchronous, the processors are labeled, etc. The present paper concentrates on the study of a new model of computation, called wireless ring.

1.1 The model

In the standard anonymous hardware ring model (cf. [3, 4, 10]) processors do not have identities. Each processor has a physical link to each of its two neighbors. There are two FIFO (first-in, first-out) buffers handling bits of the two neighboring processors (one for each neighbor). A processor may be unable to distinguish its left from its right neighbor. However each buffer of the processor is linked to a fixed neighbor. In particular, this implies that all the bits in a buffer come from the same neighbor. One distinguishes oriented and unoriented rings. In the first case buffers are labeled as L (left) and R (right) in a consistent manner for the whole network while in the second - the labeling can be done only locally, with possible inconsistencies. However, both oriented and unoriented hardware rings have the following important *separation property* which plays a crucial role in all computations: given any set of received bits, a processor can correctly partition it into subsets coming from different neighbors. Moreover, the processor can send a bit to anyone of its neighbors at will.

The situation is different in the anonymous wireless ring model. In this case each processor has only a single FIFO buffer to accomodate communication with its two neighbors. Thus a bit arriving in the buffer may come from either of the processor's neighbors although bits from a given neighbor arrive in the buffer in the order they were sent. The separation property is not valid in the present case. This is due to the fact that a processor is not aware of the physical connections to its neighbors, in fact such physical connections need not be assumed, whence the name "wireless". A bit transmitted by the processor is automatically sent from its single buffer to both neighbors.

An anonymous wireless ring may be viewed as having a higher degree of anonymity than the hardware ring. While in the latter (even for unoriented rings) a processor may at least locally label buffers (and thus corresponding neighbors), for wireless rings even local connections are anonymous for the processor. Since no physical links have to be assumed, the wireless model has potential applications in cases when communication is not performed through hardware channels, but e.g. by radio transmissions.

While boolean functions computable in a standard anonymous hardware ring are easy to determine, the computational power of wireless rings is not at all obvious. A priori, it is even not clear whether one can compute any non constant function in the anonymous wireless ring. In the sequel we will examine the behavior of synchronous and asynchronous rings. The questions that will interest us are the following: "Which boolean functions are computable in the wireless ring (synchronous or not)", "Are there any differences in the computational power of the wireless and hardware models?"

1.2 Results of the paper

The most general computational problem on a ring is input collection. In the standard asynchronous (respectively, synchronous) ring it is easily done with $O(N^2)$ bit complexity (respectively, with running time $O(N)$), where N is the size of the ring. In synchronous rings bit complexity can be reduced to $O(N \log N)$ [3] but with an exponential time-delay.

We show that the anonymous wireless and hardware rings have the same computational power, both in terms of the class of boolean functions they can compute (i.e. the class of functions which are invariant under cyclic shifts and reflections of the input) and in terms of bit complexity of input collection (i.e. total number of bits transmitted by all processors during the execution of the algorithm on any given input). For an anonymous wireless ring we give a synchronous input collection algorithm running in linear time (and thus using $O(N^2)$ bits), while in the asynchronous model we show how to perform input collection using a quadratic number of bits. Our algorithms are completely different from those used in the standard ring. However, combining our methods with ideas from [3], it is possible to perform synchronous input collection with $O(N \log N)$ bits in exponential time.

1.3 Terminology and notation

In order to define the problem we label processors with consecutive integers $0, \ldots, N-1$. It should be stressed that these labels are not known to processors and are not used in the algorithms. All arithmetic operations on integers are performed modulo N.

A sequence of bits $\beta = < b_p : 0 \leq p < N >$ is called an input sequence.

Given an input sequence β and an integer $0 \leq k \leq \lfloor \frac{N}{2} \rfloor$ we define the k-neighborhood of a processor p, denoted by $N_k(p)$, as the sequence of $2k+1$ bits $< b_i^p : -k \leq i \leq k >$ such that

$$N_k(p) = \begin{cases} < b_{p+i} : -k \leq i \leq k > & \begin{array}{l}\text{if } b_{p+i} = b_{p-i}, \text{ for all } -k \leq i \leq k, \text{ or} \\ b_{p+k_0} < b_{p-k_0}, \text{ where } -k \leq k_0 < 0 \text{ is} \\ \text{maximal s.t. } b_{p+k_0} \text{ and } b_{p-k_0} \text{ differ}\end{array} \\ < b_{p-i} : -k \leq i \leq k > & \text{otherwise.} \end{cases}$$

The pair of bits $< min(b^p_{-k}, b^p_k), max(b^p_{-k}, b^p_k) >$ is called the k-pair of the processor p, for any $0 \leq k \leq \lfloor \frac{n}{2} \rfloor$. We also define left and right k-neighborhoods of p — the sequences $< b^p_{-k}, \ldots, b^p_0 >$ and $< b^p_0, \ldots, b^p_k >$, respectively. If the neighborhood of p is not symmetric and k_0 is as above, the processor $p-1$ $(p+1)$ is called the left (right) neighbor of p if $b_{p+k_0} < b_{p-k_0}$, and the right (left) neighbor of p, otherwise.

Concatenation of two strings is denoted by $\sqcap$.

Input Collection Problem: Given an input sequence β, compute in each processor its $\lfloor \frac{N}{2} \rfloor$-neighborhood.

2 Synchronous Ring

In this section we give an input collection algorithm which is valid only for the synchronous ring. The main difference between algorithms for standard rings and for wireless ones is that in the wireless ring processors must have a mechanism allowing them to distinguish bits coming from each of the neighbors. In the standard ring this is straightforward, since bits come to different buffers. The main result of this section is

THEOREM **1** *Input collection can be done in an N-processor wireless synchronous ring in linear time with $O(N^2)$ one-bit messages.*

We start with an informal description of our algorithm. Next we describe it formally and prove its correctness. The complexity of the algorithm easily follows from its description.

The algorithm acts in $\lfloor \frac{N}{2} \rfloor$ phases numbered $0, 1, \ldots, \lfloor \frac{N}{2} \rfloor - 1$. The aim of phase k is for each processor p to compute its $(k+1)$-neighborhood. The phase consists of 24 steps, each taking constant time. The steps in the phase are divided into 6 groups, ordered and labeled as follows.

I	II	III	IV	V	VI
ε	00	0000	REPEAT	0000′	CONFLICT
	01	0001		0001′	
	11	0011		0011′	
		0100		0100′	
		0101		0101′	
		0111		0111′	
		1100		1100′	
		1101		1101′	
		1111		1111′	

The meaning of the labeling will be explained in the sequel.

At the beginning of phase k, $0 \leq k < \lfloor \frac{N}{2} \rfloor$, each processor already knows its k-neighborhood. In phase k processor p sends its k-pair to the neighbors and gets four bits A, B, C, D from them - their k-pairs. Processor p sends the k-pair in a step whose label depends on the history of computations in previous phases. Processor p uses step ε (ε denotes the empty string) until the earliest phase in which a neighbor of p reports a 'conflict'. A conflict is reported by a processor in the first phase in which pairs of bits coming from its neighbors differ. The processor reporting the conflict (in step CONFLICT) forces its neighbors to send their pairs in different steps in all subsequent phases, in order to guarantee the correct computation of the processor's neighborhood. Since p has only 2 neighbors, it can get at most 2 conflict messages, at most one from each of its neighbors. Let l be the first phase in which p gets a conflict message and let EF be its l-pair. In subsequent phases $l+1, l+2, \ldots$ processor p sends its $(l+1)$-pair, $(l+2)$-pair, etc., in steps labeled with the pair EF. If in phase l processor p gets two conflict messages from both of its neighbors then its sending step remains EF in all subsequent phases. Otherwise, it may happen that p gets the conflict message in a phase $l' > l$ from the other neighbor. Let GH be the l'-pair of p. Responding to this second conflict message, p sends its pairs in the subsequent phases in steps labeled $EFGH$, except in one case to be discussed later.

Now we describe how p processes four bits which it got in phase k. The goal is to extend the k-neighborhood $N_k(p)$ to the $(k+1)$-neighborhood $N_{k+1}(p)$. This is done in the following way. Observe that two of the four bits A, B, C, D are b^p_{-k+1} and b^p_{k-1} (say C, D) and can be discarded from further considerations. (Remark: we must be more careful if $k = 0$, but this is only a technical problem and we show how to resolve it in the formal description of the algorithm.) Two remaining bits A, B are bits of the $(k+1)$-pair of p. Now p must decide which of these two bits extends the left neighborhood and which the right one. The decision is simple if left and right k-neighborhoods of p are the same. In this case the left neighborhood is extended by $min(A, B)$ and the right one by $max(A, B)$. If left and right k-neighborhoods are not the same then p must have reported a conflict in some previous phase, say l. Let A_1, B_1, C_1, D_1 be four bits which p got in phase l. W.l.o.g. assume that $b^p_{l-1} = C_1 = b^p_{-l+1} = D_1$ and $A_1 < B_1$. Ordered pairs of bits A_1C_1 and D_1B_1 are called conflict pairs for this conflict - they are the l-pairs of p's neighbors. As described earlier, the neighbors of p send their pairs, starting from phase $l+1$, in the subsequent phases in different steps whose labels contain the conflict pairs A_1C_1 and D_1B_1. W.l.o.g. assume that bits A, B came in steps whose labels contain the conflict pairs A_1C_1 and D_1B_1, respectively. Processor p extends its left neighborhood with bit A and the right one with B.

What remains to be explained is when steps of groups IV and V are used. Let us consider 5 consecutive processors $p_{-2}, p_{-1}, p_0, p_1, p_2$ and three phases $l_1 < l_2 < l_3$ such that in phase l_1 processor p_2 reports a conflict and the conflict pair of p_1 is AB, in phase l_2 processor p_0 reports a conflict and the conflict pairs of p_{-1} and p_1 are AB and CD respectively, in phase l_3 processor p_{-2} reports a conflict and the conflict pair of p_{-1} is CD. Observe that in phase l_3+1 both p_1 and p_{-1} send their (l_3+1)-pairs in the same step labeled $ABCD$. In this case p_0 is sometimes unable to compute correctly (l_3+1)-pairs of its neighbors. Therefore p_0 sends REPEAT message in a special step labeled REPEAT. Responding to this request the processor which was involved in only one conflict before phase l_3 (processor p_{-1}) sends once more its (l_3+1)-pair in step labeled $ABCD'$ and uses this time step in all remaining phases.

Now we proceed to the formal description of the Synchronous Input Collection Algorithm. Rather than specify things to be done in each consecutive step of a phase, processor p will fix in advance the actions to be performed at a particular moment in the future during this phase.

Synchronous Input Collection Algorithm (for processor p)

$SENDING\_TIME := \varepsilon$;
/* p sends bits of its k-pair in time step labeled by $SENDING\_TIME$/*
$I\_REPORTED\_CONFLICT := FALSE$;
$b^p_0 := INPUT\_VALUE$;
for $k := 0$ **to** $\lfloor N/2 \rfloor - 1$ **do**
  $SEND(SENDING\_TIME, b^p_{-k}, b^p_k)$; /* if $k = 0$ then send 01 instead of 00 */
  if (a 1 has arrived in step $REPEAT$)
    and ($SENDING\_TIME$ was in group II in previous phase) **then**
    $SENDING\_TIME := SENDING\_TIME\sqcap'$;
    $SEND(SENDING\_TIME, b^p_{-k}, b^p_k)$
  fi;
  $EXTEND\_NEIGHBORHOOD$;
  if ($I\_REPORTED\_CONFLICT = FALSE$) and ($b^p_{-k-1} \neq b^p_{k+1}$)
  then $REPORT\_CONFLICT$;
  if (something has arrived in step $CONFLICT$)
  then $ACCEPT\_CONFLICT\_REPORT$
od

The procedures

The input collection algorithm involves several procedures which we now formally define.

procedure $SEND(TIME, A[, B])$
send bit A[and B] in step $TIME$

procedure $REPORT\_CONFLICT$
$I\_REPORTED\_CONFLICT := TRUE$;
$SEND(CONFLICT, 1)$

procedure $ACCEPT\_CONFLICT\_REPORT$
$A := \min\{b^p_{-k}, b^p_k\}$;
$B := \max\{b^p_{-k}, b^p_k\}$;
/* if $k = 0$ and $b^p_0 = 0$ then $A := 0; B := 1$*/

$SENDING\_TIME := SENDING\_TIME \sqcap A \sqcap B$

procedure $EXTEND\_NEIGHBORHOOD$
if $(I\_REPORTED\_CONFLICT = FALSE)$ **then**
  /* if $k = 0$ then decoding should be as follows. If the coming bits are
    $1111, 0011, 0111$ then $b^p_{-1} := b^p_1 := 1$,
    $b^p_{-1} := b^p_1 := 0, b^p_{-1} := 0$ and $b^p_1 := 1$, resp. */
  delete two bits equal to b^p_{-k+1} and b^p_{k-1} from four
  bits that came in steps of the groups I or II;
  b^p_{-k-1} := smaller one of remaining two bits (any if they are equal);
  b^p_{k+1} := the other bit
else
  $TIME_2 := OLDER$;
  /* If before the phase in which processor p reported a conflict,
  exactly one of p's neighbors had been involved in another conflict,
  then OLDER returns $SENDING\_TIME$ label of this neighbor,
  otherwise $OLDER$ returns the label of the latest step in which p received a k-pair */
  if (both k-pairs arrived in $TIME_2$) **then**
    $SEND(REPEAT, 1)$;
    $TIME_1 = TIME_2 \sqcap'$;
    delete two bits equal to those that arrived in $TIME_1$
    from bits that arrived in $TIME_2$
  else $TIME_1$:= the last step different from $TIME_2$
            in which a k-pair arrived;
  if $IS\_LESS(TIME_1, TIME_2)$ **then**
    delete a bit equal to b^p_{k-1} from the bits that came in $TIME_1$;
    b^p_{-k-1} := the remaining bit;
    delete a bit equal to b^p_{-k+1} from the bits that came in $TIME_2$;
    b^p_{k+1} := the remaining bit
  else
    delete a bit equal to b^p_{k-1} from the bits that came in $TIME_2$;
    b^p_{-k-1} := the remaining bit;
    delete a bit equal to b^p_{-k+1} from the bits that came in $TIME_1$;
    b^p_{k+1} := the remaining bit
  fi
fi

function $OLDER$

Let l be the index of the phase in which p reported a conflict;
if (in phase l $SENDING_TIME$s of both neighbors were ε) **then**
  $OLDER :=$ the label of the step in the current phase
    in which the last k-pair arrived
else
  if (in phase $l+1$ at least one $(l+1)$-pair arrived
    in some step from group V) **then**
    $OLDER :=$ the label of the last such step
  else
    $OLDER :=$ the label of the last step from group III in phase $l+1$
      in which a $(l+1)$-pair arrived
    fi
fi

By $PREFIX$ (respectively, $SUFFIX$) of a string we will always mean its prefix (respectively, suffix) of length 2 (the character $'$ does not count, so that $SUFFIX(ABCD') = CD$). Strings are compared lexicographically.

function $IS\_LESS(TIME_1, TIME_2)$
Let l be the index of the phase just after the phase in which p reported conflict.
if both l-pairs arrived in steps of the group II **then**
  $IS\_LESS := [PREFIX(TIME_1) < PREFIX(TIME_2)]$
fi
if both l-pairs arrived in steps of the the group III **then**
  $IS\_LESS := [SUFFIX(TIME_1) < SUFFIX(TIME_2)]$
fi
if one l-pair arrived in a step of the group II
  and the other in a step of the group III **then**
  $IS\_LESS := [PREFIX(TIME_1) < SUFFIX(TIME_2)]$
fi

EXAMPLE **1** The following table shows an example of running the above algorithm on the 10-processor ring.

Proc.	0	1	2	3	4	5	6	7	8	9
Input	0	1	1	1	1	1	1	1	1	0
Phase	Sending Times									
0	ε $\mathcal{C}$	ε $\mathcal{C}$	ε	ε	ε	ε	ε	ε	ε $\mathcal{C}$	ε $\mathcal{C}$
1	01	11	11 $\mathcal{C}$	ε	ε	ε	ε	11 $\mathcal{C}$	11	01
2	01	1101	11	11 $\mathcal{C}$	ε	ε	11 $\mathcal{C}$	11	1101	01
3	01	1101	1101	11	11 $\mathcal{C}$	11 $\mathcal{C}$	11	1101	1101	01
4	01	1101	1101 $\mathcal{R}$	1101 $1101'$	1111	1111	1101 $1101'$	1101 $\mathcal{R}$	1101	01

The rows of the table correspond to phases of the algorithm, the columns correspond to processors. The entry in row k and column p is the SENDING_TIME label of processor p in phase k (the step in phase k in which p's neighbors receive its neighborhood information). The letter $\mathcal{C}$ in the entry of the table means, that the corresponding processor reports conflict, the letter $\mathcal{R}$ means, that it sends a REPEAT request in the current phase.

Let us trace the execution history for processor 2. In phase 0, in step ε it sends 11 and receives four 1s from its neighbors. Processor 1 reports conflict, so 2 changes its SENDING_TIME to 11 (the bits it has just sent); however, its 1-neighborhood - 111 - is still symmetric. In phase 1 it sends its 1-pair: 11, and receives four bits: 0111. The bits from processor 1 and 3 arrive in different steps, but 2 doesn't make use of it yet. After erasing bits from its 0-pair, the remaining bits are 0 and 1, so the 2-neighborhood 01111 is not symmetric any more and processor 2 reports conflict. ¿From now on processor 2 distinguishes its left neighbor: the one which sent 01, and its right neighbor: the one which sent 11. In phase 2 (in which it sends 01, still in step 11) it gets: 01 in step 1101 and 11 in step 11. The conflict pairs are: SUFFIX of the longer label (which belongs to processor 1; its PREFIX is used by processor 0 to resolve its conflict from phase 0) and PREFIX of the shorter label (which at the moment happens to be the whole label). By comparing labels of arrival steps according to the conflict pairs, processor 2 can deduce, that 01 is from its left neighbor, and 11 from the right neighbor, hence the 3- neighborhood is 0011111, not 1011110. At the end of phase 2 another conflict report arrives, so 2 extends its SENDING_TIME with the pair of bits it sent in this phase, hence the label becomes 1101. In phase 3 processor 2 sends 01 in step 1101 and receives: 11 in step 1101 and 01 in step 11. Decoding is the same, as previously, and the 4-neighborhood is 100111111. Also in this phase processor 3 accepts conflict report from processor 4 and changes its SENDING_TIME to 1101 - the same, as processor 1, so in phase 4 processor 2 gets all four bits 0111 in one step 1101. It sends the REPEAT request and gets two bits - 01 in step $1101'$ from 3 (not 1, because it didn't accept conflict in the previous phase). Processor 2 knows now, that 11 is from the neighbor which was involved in two conflicts by phase 3 (it happens to be the left neighbor), and 01 from the neighbor, which was involved in one conflict by phase 3 (the right one). Hence its (whole) neighborhood is 11001111111 (the first and the last 1 are in fact the input value of processor 7 - opposite to 2 on the ring).

Proof of Theorem 1

It is enough to prove the correctness of our algorithm. Its complexity easily follows from the description: each execution of the **for** loop takes constant time. We will show that in phase k each processor correctly computes its $(k+1)$-neighborhood. We consider the following two cases.
CASE 1: $I\_REPORTED\_CONFLICT = FALSE$.
We will need the following lemma.

LEMMA **1** *$I\_REPORTED\_CONFLICT = FALSE$ in the beginning of phase k if and only if $b^p_{-l} = b^p_l$, for all $l \leq k$.*

PROOF. Observe that the $I\_REPORTED\_CONFLICT$ flag is set just in case the neighborhood of a processor stops being symmetric . ■

In this case decoding is easy. The four bits that come from neighbors (possibly even all four in one step) are $b^p_{-k-1}, b^p_{-k+1}, b^p_{k-1}, b^p_{k+1}$. (If $k = 0$ then these bits are $b^p_{-1}, 1, 1, b^p_1$.) Processor p already knows b^p_{-k+1}, b^p_{k-1}. So b^p_{-k-1} and b^p_{k+1} constitute the remaining pair. If they are not equal then the processor must $REPORT\_CONFLICT$ and from now on it will have to distinguish the neighborhood information sent by its neighbors: the left (the one who sent b^p_{-k-1} and b^p_{k-1}) and the right (the one who sent b^p_{-k+1} and b^p_{k+1}).
CASE 2: $I\_REPORTED\_CONFLICT = TRUE$.
Denote by l the phase in which p reported conflict. The distinction mentioned above is made by comparing arrival times of the information. If

some neighbor of the processor has reported conflict, it extends its SENDING_TIME label adding the pair of bits it has just sent in non-decreasing order.

LEMMA **2** *If a processor has reported conflict then the conflict pairs of the neighbors are either (*00 *for the left and* 01 *for the right) or (*01 *for the left and* 11 *for the right).*

PROOF. By lemma 1 the conflict pairs must have one common bit $b^p_{-k+1} = b^p_{k-1}$ (for $k = 0$ the common bit equals 1) but they cannot be the same, because there was a conflict. ■

If pairs of bits have arrived in different steps, processor p can decide which pair is from its left neighbor and which from the right one. The decision depends on positions of conflict pairs in $SENDING\_TIME$ labels of its neighbors. If in phase l both labels were ε then the conflict pairs are PREFIXes of the labels. Similarily, if in phase l both $SENDING_TIME$s were in group II, then the conflict pairs are SUFFIXes of the labels. If in phase l exactly one neighbor q of p had $SENDING\_TIME$ in group II, then p should compare SUFFIX of q's current $SENDING\_TIME$ (computed by the function OLDER) with PREFIX of the other neighbor's $SENDING\_TIME$ (cf. definition of the function IS_LESS).

The rest of the proof is based on the following lemma.

LEMMA **3** *Let l be a phase in which processor p reports a conflict. Then in each subsequent phase $k > l$ SENDING_TIMEs of its neighbors differ, except in at most one phase in which p sends the $REPEAT$ request.*

PROOF. Consider two cases.

a) In phase l SENDING_TIMEs of p's neighbors have the same lengths.

In this phase both neighbors of p extend their SENDING_TIMEs with their conflict pairs (which are different by lemma 2). Since conflict pairs occupy the same positions in both SENDING_TIMEs, the conclusion follows.

b) In phase l SENDING_TIMEs of p's neighbors have different lengths.

It follows from the description of the algorithm, that one of the neighbors (say p') sends its l-pair in step labeled ε and the other (say p'') in a step of the group II (say AB). Let CD, EF be conflict pairs in p's conflict, for p' and p'' respectively. If $CD \neq AB$ then in the subsequent phases SENDING_TIMEs of p' and p'' always differ.

Assume $CD = AB$. There are two subcases.

b1) In subsequent phases p' is never involved in another conflict.

Observe that in this case SENDING_TIMEs of p' and p'' have different lengths until the end of the algorithm.

b2) Let $k > l$ be a phase in which p' is involved in a conflict reported by its neighbor different from p. Let GH be a conflict pair of p' in phase k. If $GH \neq EF$ then the conclusion follows immediately. Assume $GH = EF$. It follows from the description of the algorithm, that p' can change its SENDING_TIME $CDGH$ into $CDGH'$ only in phase $k+1$, responding to the REPEAT message of p. Processor p sends such a message only if SENDING_TIMEs of its neighbors are the same in phase $k+1$. To finish the proof of the lemma observe that the only phase in which p'' can change its SENDING_TIME $ABEF$ into $ABEF'$ is the phase $l+1$. However in this case SENDING_TIMEs of p' and p'' are different in phase $k+1$, so p does not send the REPEAT message. ■

If processor p sends the REPEAT message in phase k then exactly one of its neighbors sends again its k-pair (the one which had received no conflict reports before p reported conflict). Since only one processor repeats its message and in subsequent phases SENDING_TIMEs of p's neighbors are different, p can always correctly extend its neighborhood. This completes the proof of Theorem 1. ■

3 Asynchronous Ring

In this section we focus on the case of asynchronous wireless rings. In the asynchronous model processors may remain idle for arbitrary finite time. Since a processor has only a single buffer it seems possible that information contained in it comes from one neighbor while the other remains idle. Hence, a priori, it is not obvious how to perform any non-trivial calculation. Here we will prove that in fact the asynchronous model has the same computational power as the synchronous model. We show a general simulation scheme which converts any synchronous algorithm working on a ring in time t into an asynchronous algorithm in which every processor sends $O(t)$ bits. (Schemes of this type are called synchronizers (cf. [5]).)

Consider a synchronous algorithm working in t time steps. In step k of the algorithm the processor performs some action A_k (depending not only on k, but on the whole previous execution history). There are three possible actions: 'send 0', 'send 1', and 'send nothing'.

Every processor in the asynchronous algorithm works in t phases. In phase k the processor sends information about the action A_k to both its neighbors. After receiving information about actions performed by its neighbors in step k, it updates the execution history and is able to decide about the action A_{k+1}.

The encoding of information must be done in such a way that the processor be able to

- recognize that all information from phase k has already arrived,
- separate this information from information sent by its neighbor(s) in a different phase(s),
- decode both its neighbors' actions from step k.

For action A_k define $\alpha(A_k)$:

$$\alpha(A_k) = \begin{cases} a_1 & \text{if } k \text{ is odd and } A_k \text{ is 'send 0'} \\ a_2 & \text{if } k \text{ is odd and } A_k \text{ is 'send 1'} \\ a_3 & \text{if } k \text{ is odd and } A_k \text{ is 'send nothing'} \\ a_4 & \text{if } k \text{ is even and } A_k \text{ is 'send 0'} \\ a_5 & \text{if } k \text{ is even and } A_k \text{ is 'send 1'} \\ a_6 & \text{if } k \text{ is even and } A_k \text{ is 'send nothing'} \end{cases}$$

where $a_i = 2^i - 1$, for $i = 1, \ldots, 6$.

Asynchronous Simulation Scheme (for a processor p)
for $k = 1$ **to** t **do**

1. send $(0^{\alpha(A_k)}1^{\alpha(A_k)})$;
2. wait until there are as many 1s as 0s in the buffer,
   and the number of 0s is of the form $a_i + a_j + c$, where
   $i, j \in \{1, 2, 3\}, c = 0$ or $c \geq a_4$ if k is odd,
   $i, j \in \{4, 5, 6\}, c < a_4$ if k is even;
3. decode neighbors' actions according to the list above (one neighbor's action is the one corresponding to a_i, the other's - to a_j);
4. update the synchronous algorithm's execution history and
   calculate the action A_{k+1};
5. update the buffer, that is delete $a_i + a_j$ 0s and the
   same number of 1s

od

THEOREM **2** *For every synchronous algorithm running on a wireless ring R in t time units there exists an asynchronous algorithm computing the same function on R, in which each processor sends $O(t)$ bits.*

PROOF. It is enough to prove that our Asynchronous Simulation Scheme works correctly. The complexity follows from the fact, that in each execution of the **for** loop every processor sends a constant number of bits. We prove that what processor p decodes in phase k is exactly what its neighbors have sent. It is easy to see that

LEMMA **4** *If a number can be represented as a sum of at most two (possibly the same) elements of the set* $\{a_1, a_2, a_3\}$ *and at most two (possibly the same) elements of* $\{a_4, a_5, a_6\}$ *then this representation is unique.* ■

We say that the message sent by a processor to its neighbor p in some phase k is "on the way" to p, if some nonempty part of it has already arrived to p, but the rest (also nonempty) is still delayed.

LEMMA **5** *The number of* 1*s in* p*'s buffer is equal to the number of* 0*s if and only if no message is "on the way" to* p*.*

PROOF. The conclusion follows from the fact that 0s are always sent before 1s and bits come in order they were sent. ■

Now we prove that the above algorithm is "almost synchronous":

LEMMA **6** *If a processor is in phase* k*, then none of its neighbors has completed step* 2 *in its phase* $k+1$*.*

PROOF. Let us consider the first (according to some particular execution) situation such that processor p has completed step 2 in its phase $k+1$, while one of its neighbors is still in phase k. As decoding has been correct so far, p has successfully deleted from its buffer all information from earlier phases (up to k), and the content of its buffer must be from its other neighbor's phases $k+1$ and possibly $k+2$ (not $k+3$ or higher since we are considering the first such situation). However, by lemma 5 if the number of 1s equals the number of 0s then no message can be 'on the way' to p. Thus by lemma 4, p can recognize that the number of 0s contains only one number from the apropriate set ($\{a_1, a_2, a_3\}$ if $k+1$ is odd, $\{a_4, a_5, a_6\}$ otherwise) and consequently p cannot have completed step 2, which is a contradiction. ■

Now decoding is easy: a processor completes step 2 in its phase k when the number of 0s in the buffer is the sum of exactly two elements of the apropriate set ($\{a_1, a_2, a_3\}$ if k is odd, $\{a_4, a_5, a_6\}$ otherwise) and at most two elements of the other set. Lemma 4 states, that the correct decoding can be achieved. ■

The above theorem implies the following.

COROLLARY **1** *Input collection can be done in an N-processor wireless asynchronous ring using $O(N^2)$ one-bit messages.* ■

4 Conclusion

In this paper we have discussed the problem of computing on anonymous wireless rings. We gave algorithms for input collection, which unlike in the case of the standard hardware model, are quite non-trivial. However, both in the synchronous and asynchronous cases, the complexity of input collection turned out to be the same as in the standard model ($O(N)$ time in the synchronous case and $O(N^2)$ bits in the asynchronous case). Combining our methods with those from [3] it is possible to reduce bit complexity in the synchronous case to $O(N \log N)$ but with exponential time delay. Since input collection is the most general computational problen on the ring, our results show that all boolean functions invariant under cyclic shifts and reflections of the input can be efficiently computed on an anonymous wireless ring. This class of functions is identical to the class of functions computable on an anonymous hardware ring. Thus, in spite of much weaker assumptions of our model its computational performance is the same as of the classical one.

There are several open problems in the wireless ring suggested by studies of the hardware model, like symmetry breaking [12], extrema finding [7], language complexity [4], leader election [9], etc. Another interesting topic worth investigating concerns input collection algorithms in other types of anonymous wireless networks (e.g. tori, hypercubes or arbitrary networks). We do not know if input collection is at all possible in arbitrary wireless networks and if so, how efficiently it can be performed.

Acknowledgements

Many thanks to Danny Krizanc for useful conversations.

References

[1] K. Abrahamson, A. Adler, R. Gelbart, L. Higham and D. Kirkpatrick, “The Bit Complexity of Randomized Leader Election on a Ring”, SIAM Journal on Computing, 18(1): 12 - 29, 1989.

[2] K. Abrahamson, A. Adler, R. Gelbart, L. Higham and D. Kirkpatrick, "Randomized Function Evaluation on a Ring", Distributed Computing, 3(3): 107 - 117, 1989.

[3] H. Attiya, M. Snir and M. Warmuth, "Computing on an Anonymous Ring", Journal of the ACM, 35 (4), 1988. (Short version has appeared in proceedings of the 4th Annual ACM Symposium on Principles of Distributed Computing, 1985, 845 - 875.)

[4] H. Attiya and Y. Mansour, "Language Complexity on the Synchronous Anonymous Ring", Theoretical Computer Science, 53 (1987) 169 - 185.

[5] B. Awerbuch, "Complexity of Network Synchronization", JACM, 34, 4, 804 - 823 (1985).

[6] H. L. Bodlaender and G. Tel, "Bit Optimal Election in Synchronous Rings", Information Processing Letters, 36 (1990) 53 - 56.

[7] E. Chang and R. Roberts, "An Improved Algorithm for Decentralized Extrema Finding in Circular Arrangements of Processors", Comm. ACM 22 (1979) 281 - 283.

[8] D. Dolev, M. Klawe and N. Rodeh, "An $O(n \log n)$ Unidirectional Algorithm for Extrema Finding in a Circle", J. Algorithms 3 (1982) 245 - 260.

[9] G. N. Frederickson and N. A. Lynch, "Electing a Leader in a ring", J. ACM 34 (1987) 95 - 115.

[10] S. Moran and M. K. Warmuth, "Gap Theorems for Distributed Computation", in Proc. 5th ACM Symp. on Principles of Distributed Computing (1986) 131 - 140.

[11] G. L. Peterson, "An $O(n \log n)$ Unidirectional Algorithm for the Circular Extrema problem", ACM Trans. on Prog. Lang. and Systems 4(1982) 758 - 762.

[12] P. Spirakis, V. Tampakas and A. Tsiolis, "Symmetry Breaking in Asynchronous Rings with $O(n)$ Messages", in: J.-C. Bermond, M. Raynal (eds.), Distributed Algorithms: 3rd International Workshop, LNCS 392, Springer Verlag, 1989, pp. 233 - 241.