SIMULATION OF PARALLEL ALGORITHMS ON A DISTRIBUTED NETWORK

Anneke A. Schoone and Jan van Leeuwen

RUU-CS-86-1
January 1986

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

SIMULATION OF PARALLEL ALGORITHMS ON A DISTRIBUTED NETWORK

Anneke A. Schoone and Jan van Leeuwen

Technical Report RUU-CS-86-1
January 1986

Department of Computer Science
University of Utrecht
P.O. Box 80.012
3508 TA Utrecht
the Netherlands

SIMULATION OF PARALLEL ALGORITHMS ON A DISTRIBUTED NETWORK

Anneke A. Schoone and Jan van Leeuwen

Department of Computer Science, University of Utrecht,
P.O.Box 80.012, 3508 TA Utrecht, the Netherlands.

Abstract. We investigate the possibilities to ensure synchronization in a distributed network by enhancing the link-level protocol used for communication on the network links. It is shown that distributed (local) synchronization and link-level message control can be achieved by means of one protocol skeleton, that can be proved correct by the technique of system-wide invariants. The result is used to obtain a general protocol for the message-efficient simulation of (synchronous) parallel algorithms on (asynchronous) distributed networks that requires only one bit of extra information in every message for control purposes in the simplest case.

1. Introduction. Since it is usually easier to design and analyze a parallel algorithm than a distributed algorithm, it would be helpful to have a standard transformation from the one type of algorithm to the other. Any general transformation technique of this sort will allow programmers to write a distributed (hence, asynchronous) algorithm as if it were run on a synchronous network. Awerbuch [1] refers to such transformations as "synchronizers", and has shown the existence and use of rather efficient synchronizers. His technique forces a more or less system-wide synchronization of the distributed (i.e. asynchronous) network, at the expense of a small number of extra control messages.

In this paper we explore a different and more pragmatic approach to synchronization in distributed networks. First, we only strive for synchronization between processors in the network that communicate directly over a link. After all, the question whether processors that do not directly communicate are completely synchronized is irrelevant to the correctness of a parallel algorithm. Second, we note that in most distributed networks there are so-called link-level protocols (like the alternating bit and two-way sliding window protocols) in effect that

control the exchange of messages between neighboring processors and guard against the loss of information, and that are based on a weak form of synchronization between sender and receiver. As the synchronizing tool may thus exist already at the level of current protocols, we will attempt to exploit the type of synchronization that is inherent to several link-level protocols to ensure (local) synchronization. This is done to minimize the amount of overhead.

We show that distributed (local) synchronization and link-level message control can be achieved by means of one protocol skeleton, that can be tuned to one's further needs. To prove the partial correctness of the general protocol skeleton and derive results about the degree of synchronization in the network, we use a technique due to Krogdahl [3] and Knuth [2] using system-wide invariants. The synchronizing protocol actually is an extension of the general approach to link-level protocols presented in [4] and [5], where it was shown that the alternating bit protocol and the "balanced" two-way sliding window protocol are instances of one general protocol skeleton, and can thus be treated together.

The remainder of the paper is organized as follows. Section 2 contains the assumptions about the network model and the definition of the general protocol skeleton, following the style of [5]. Section 3 contains a first version of the protocol skeleton of the general synchronizer, of which we show that the restriction to the practical case of the alternating bit protocol can give rise to deadlock. In section 4 we present a modification of the synchronizer skeleton which can use alternating bit protocols and is free of deadlock in all cases. Section 5 contains a synchronizer which achieves the same degree of synchronization as the alternating bit synchronizer of section 4, but which is free from the basic asymmetry of the latter (i.e., the asymmetry of "who begins" in the protocol). In section 6 we discuss the composition (concatenation) of protocols, and in section 7 we offer some final comments.

2. Preliminaries. Our approach is motivated by the wish to simulate parallel algorithms, i.e., synchronous computations, on a distributed network that has the right structure but not necessarily the required

synchronous mode of processor activity. The model we use for parallel computations is the following. Characteristics of a parallel algorithm are:

P1. N processors (nodes) in a fixed interconnection network.

P2. Every processor executes the same algorithm but uses (perhaps) different data.

P3. The processors execute in a synchronized manner, i.e., they proceed in lock-step fashion.

We will assume that each processor in a parallel algorithm performs a cycle of the type

```
Cp. while no termination do
    begin communicate with all neighbors,
          i.e., send to and receive from them;
          compute
    end.
```

(without loss of generality). Communication between processors consists of a send and a receive over the connecting link, i.e., an exchange of one packet of information between neighboring processors. This leads to another characteristic:

P4. Every processor communicates with all its neighbors in one time step.

The latter characteristic is the reason for wanting processor networks in which the nodes have low degrees. We assume that the communication subsystem is fault and error free. Complexity is measured in terms of parallel time.

The model we use for distributed computations is fairly standard also. Characteristics of a distributed algorithm are:

D1. N processors in a (fixed) interconnection network.

D2. Every processor uses the same communication primitives but uses (perhaps) a different algorithm and different data.

D3. Every processor executes at its own finite (non-zero) speed and all links carry packets at their own finite (non-zero) speed.

We will assume that each processor in a distributed algorithm performs a cycle of the type

```
Cd. while no termination do
    begin communicate with a neighbor,
          i.e., send to or receive from it;
          compute
    end.
```

(without loss of generality). We do not require that data is sent to or received from all neighbors in a cycle, and assume the entire operation is asynchronous. We assume that the cycle specifies either a send or a receive when the cycle gets to the communication statement in the cycle, e.g., as in CSP. Additional characteristics are:

D4. Messages may be lost or damaged in a link.

D5. Links may go down or come up.

D6. Processors may go down or come up.

D7. Every processor knows its neighbors.

D8. Every processor proceeds with its algorithm in some finite time (unless it crashes).

D9. Links preserve the ordering of messages sent on that link.

Complexity is measured in terms of the number of messages or bits exchanged.

To control and deal with the possible loss or damage of messages (D4) there exist several link-level protocols, e.g., the commercial "embedded" versions of the alternating bit and sliding window protocols. We will assume that networks have low degree and that the processors exchange messages according to some version of the alternating bit or sliding window protocol. Hence we assume that the communication can be described by the unifying protocol skeleton presented in [4] and [5]. As we will build on it in the remainder of this paper, we need the terminology and the definition of the protocol skeleton.

Essential in our approach is the idea that processors will generate sequences of messages that need to be exchanged. The generation process may depend on past messages, as will be the case in the simulation of

parallel algorithms. For the time being we will assume that the messages generated by a processor P and destined for a neighbor Q carry unique and unbounded sequence numbers and can be referred to as M_0, M_1, M_2, ... The corresponding sequence of messages generated by Q and destined for P will be referred to as N_0, N_1, N_2, ... We assume that incoming messages are buffered at either end of the link in FIFO-queues. It should be clear what is meant by the "P-to-Q queue" in this respect. We shall assume that "sending" means either inserting a message at the rear of the queue, or losing it. "Receiving" is done only when the queue is nonempty, in which case the message at the front of the queue is read and deleted from the queue.

Notation: we will add the name of a processor P as a (first) subscript to the names of variables and operations if these variables and operations are maintained in P or performed at P, respectively. We will add the name of a neighboring processor Q as a (second) subscript if the variables or operations refer to P's link to Q.

We need the following variables for a proper implementation of the protocol skeleton for two-way link-level protocols:

A_{PQ} = the number of consecutive messages that P is sure Q has received and stored,

a_{PQ} = the number of consecutive messages that P has received from Q and stored.

Here it is assumed that messages are ordered in the way they are generated by the algorithm that runs in P. In addition, for every P and Q there are two nonnegative values f_{PQ} and f_{QP}, such that $f_{PQ}+f_{QP}\geq 1$ with the following meaning: f_{QP} is a constant such that if P sends Q a message M_j, its sequence number (j) is used as an implicit acknowledgement for message $N_{j-f_{QP}}$ from Q to P. Likewise, f_{PQ} is a constant such that if Q sends P a message N_i, its sequence number (i) is used as an implicit acknowledgement for message $M_{i-f_{PQ}}$ from P to Q. Now we can formulate the possible operations $S1_{PQ}$ and $R1_{PQ}$ for P by means of which it can communicate with a neighbor Q:

$S1_{PQ}$: send message M_j, where $\min(A_{PQ}, a_{PQ}+f_{QP}-1) \leq j < a_{PQ}+f_{QP}$.

$R1_{PQ}$: receive message N_i and optionally store it;
if $i-f_{PQ}+1 > A_{PQ}$ **then** $A_{PQ} := i-f_{PQ}+1$;
a_{PQ}:= new (maximal) value such that P has received and stored messages $N_0, N_1, \ldots, N_{a_{PQ}-1}$ from Q.

Likewise, Q's communication to P is built up from the corresponding operations $S1_{QP}$ and $R1_{QP}$. Initially, we set $a_{PQ}=0$ and $A_{PQ}=-f_{PQ}$ for all P and Q.

The idea is that two processors P and Q can execute S1 and R1 actions as often as they want and in any order, and that despite this certain "system-wide" invariants about the messages transfer are maintained. The "option" in the R1 action reflects a choice which need not be made here, namely what to do with messages that arrive out of order. One possibility is to buffer these messages, another possibility is to discard them and wait for a retransmission. By a suitable choice of actions (like alternating send and receive actions whenever possible) one can try to arrive at a protocol that guarantees the transfer of the complete message-sequences, under the additional assumption that every message that is sent will arrive at least once free of error and thus be stored. For a better understanding we recall these system-wide invariants.

Lemma 2.1. Let processors P and Q be neighbors. Then

(i) A_{PQ} and a_{PQ} are nondecreasing.

(ii) $A_{PQ} \geq a_{PQ}-f_{PQ}$.

(iii) The number of possible values for j in $S1_{PQ}$ is at least 1 and at most $f_{PQ}+f_{QP}$.

(iv) $A_{PQ} \leq a_{QP}$.

(v) Let $M_{j_1}, \ldots, M_{j_r}$ be the contents of the P-to-Q queue from the front to the rear, where $r \geq 0$, and let j_{max} be the maximum index of any message that has ever been removed from the P-to-Q queue. (If nothing has ever been removed, let $j_{max}=-1$.) Let $j_0=j_{max}$ and $j_{r+1}=a_{PQ}-f_{PQ}$, then

$$j_k < j_{k'}+f_{PQ}+f_{QP} \text{ for } 0 \leq k < k' \leq r+1.$$

Of course the corresponding invariants with "P" and "Q" interchanged are valid too. For further details we refer to [4] and [5]. It can be shown that unbounded sequence numbers can always be avoided while retaining the characteristics of the protocol, and that it is sufficient in fact to count sequence numbers modulo $2(f_{PQ}+f_{QP})$. We can formulate the programcycle of a processor P as follows:

$C1_P$: **while** no termination **do**
begin do operation $S1_{PQ}$ or $R1_{PQ}$ for some neighbor Q;
compute
end.

3. The general synchronizer (first version). The partial synchronization between the two ends of every single link as implied by the specification of the link-level protocol is not sufficient for achieving local synchronization. To achieve it we have to incorporate some kind of synchronization between all link ends at every single processor. One possible adaptation of the protocol skeleton immediately comes to mind: maintain a "state counter" in every processor P that checks the atomic progress on the links incident to P and restrict for each link the sending possibilities or the receiving possibilities with respect to the value of this state counter. We will first explore the effect of restricting only the sending possibilities. In addition to the variables a_{PQ} for each link, each processor P will now also maintain a counter a_P. Initially, a_P is set to 0. We change the sending operation of processor P in the protocol skeleton as follows:

$S2_{PQ}$: send message M_j, where $\min(A_{PQ}, a_P+f_{QP}-1) \leq j < a_P+f_{QP}$.

The receive action simply remains identical to $R1_{PQ}$. However, the programcycle of processor P is changed, to implement the synchronization of the link-ends incident to P and maintain the value of a_P as follows:

$C2_P$: **while** no termination **do**
begin do operation $S2_{PQ}$ or $R1_{PQ}$ for some neighbor Q;
if for all neighbors Q $a_{PQ}>a_P$
then begin $a_P:=a_P+1$; compute **end**
end.

Hence the "compute" part of the cycle is only performed when all links have advanced far enough with their communication. To evaluate the effect of this change, we need relations which remain invariant under the operations $S2_{PQ}$, $R1_{PQ}$, $S2_{QP}$, $R1_{QP}$, $C2_P$ and $C2_Q$.

Lemma 3.1. Let processors P and Q be neighbors. Then

(i) A_{PQ}, a_{PQ} and a_P are nondecreasing.

(ii) $A_{PQ}\geq a_{PQ}-f_{PQ}\geq a_P-f_{PQ}$.

(iii) The number of possible values for j in $S2_{PQ}$ is at least 1 and at most $f_{PQ}+f_{QP}$.

(iv) $A_{PQ}\leq a_{QP}$.

(v) Let M_{j_1}, ..., M_{j_r} be the contents of the P-to-Q queue from the front to the rear, where $r\geq 0$, and let j_{max} be the maximum index of any message that has ever been removed from the P-to-Q queue. (If nothing has ever been removed, let $j_{max}=-1$.) Let $j_0=j_{max}$ and $j_{r+1}=a_P-f_{PQ}$, then

$$j_k < j_{k'}+f_{PQ}+f_{QP} \text{ for } 0\leq k<k'\leq r+1.$$

Proof. Note that $a_P\leq a_{PQ}$, from the specification of $C2_P$. The remaining proofs are completely analogous to those of the corresponding lemmas in [5]. Q.E.D.

Theorem 3.1. $a_Q-f_{QP} \leq a_P \leq a_Q+f_{PQ}$.

Proof. Have a closer look at the relationship between A_{QP} and j_{max} from lemma 3.1.(v). Initially, $j_{max}=-1$ and $A_{QP}=-f_{QP}$. Later $j_{max}=\max\{j \mid M_j$ was removed from the P-to-Q queue$\}$ and $A_{QP}=\max\{j-f_{QP}+1 \mid M_j$ was removed from the P-to-Q queue$\}$. Hence $j_{max}=A_{QP}+f_{QP}-1$. Thus by lemma 3.1.(v) $j_{max}= A_{QP}+f_{QP}-1 < j_{r+1}+f_{PQ}+f_{QP} =$

$a_P - f_{PQ} + f_{PQ} + f_{QP} = a_P + f_{QP}$. Hence $A_{QP} \leq a_P$. Now $a_Q \leq a_{QP} \leq A_{QP} + f_{QP} \leq a_P + f_{QP}$. Thus also $a_P \leq a_Q + f_{PQ}$ and the desired inequalities follow. Q.E.D.

Theorem 3.2. If M_j is in the P to Q queue, we have

$$a_Q - f_{QP} - f_{PQ} \leq a_{QP} - f_{QP} - f_{PQ} \leq j < a_Q + f_{QP} + f_{PQ} \leq a_{QP} + f_{QP} + f_{PQ}.$$

Proof. By lemma 3.1.(v) $j > j_{max} - f_{QP} - f_{PQ}$. Since $j_{max} = A_{QP} + f_{QP} - 1$ we have $j > A_{QP} - f_{QP} - 1 \geq a_{QP} - f_{QP} - f_{PQ} - 1$ by lemma 3.1.(ii). Hence $j \geq a_{QP} - f_{QP} - f_{PQ}$. Lemma 3.1.(v) gives us $j < j_{r+1} + f_{QP} + f_{PQ} = a_P - f_{PQ} + f_{QP} + f_{PQ} = a_P + f_{QP}$. By theorem 3.1 we have $j < a_Q + f_{QP} + f_{PQ}$. Further application of lemma 3.1.(ii) gives us the desired result. Q.E.D.

It follows from theorem 3.2 that it is not necessary to use unbounded sequence numbers and counters, but that it is sufficient to use sequence numbers mod n (i.e., windows of size n/2) instead, where n is any fixed number with $n \geq 2(f_{PQ} + f_{QP})$ for all links (P,Q).

For the simulation of a parallel algorithm on a distributed network, it is desirable from the efficiency point of view to have only one round of communication separated from the next round by a computation. Hence we have to restrict the sending windows to one value. Thus we want to set $f_{PQ} + f_{QP} = 1$ for all links (P,Q), and in fact we are using the alternating bit protocol (cf. [4], [5]).

Theorem 3.3. When the alternating bit protocol is used on all links and programcycle C2 in all nodes, deadlock can arise.

Proof. Consider the network consisting of three nodes P, Q and R, and the links (P,Q), (Q,R) and (R,P). Take $f_{PQ} = f_{QR} = f_{RP} = 0$ and $f_{QP} = f_{PR} = f_{RQ} = 1$. Let a_P be x. By theorem 3.1 we have $a_P \leq a_Q \leq a_P + 1$. Hence $a_Q = x$ or $a_Q = x+1$. Also $a_P - 1 \leq a_R \leq a_P$. Thus $a_R = x-1$ or $a_R = x$.

Case 1. $a_Q = x+1$. Then, as $a_Q \leq a_R \leq a_Q + 1$, $a_R \geq x+1$. Contradiction.

Case 2. $a_Q = x$. Then $a_R = x$ or $a_R = x+1$ and, hence, necessarily $a_R = x$.

Thus always $a_P = a_Q = a_R$. But this means no processor can increase its value of a_P, and hence no processor can make progress from the initial value of 0. Q.E.D.

Observe that no deadlock arises in the network in the given proof if we had taken $f_{PQ}=f_{PR}=f_{QR}=1$ and $f_{QP}=f_{RP}=f_{RQ}=0$. In the next section we modify the protocol skeleton further to avoid the possibility of deadlock, without losing the essence of the invariants and the synchronizing character.

4. The alternating bit synchronizer. The reason for deadlock in the above synchronizer was that the restriction on the sending capability of the processors, while reasonable, can be too strict. In this section we will show that the restriction can be relaxed without losing the synchronizing character of the protocol skeleton. The resulting protocol skeleton will have the desired properties of a general synchronizer. We prove this only for the case of the "augmented" alternating bit protocol, and defer the general case to section 5.

Definition. $f_P=\max\{f_{QP} \mid Q \text{ a neighbor of } P\}$.

We change the sending operation and the programcycle of every processor P as follows:

$S3_{PQ}$: send message M_j, where
$\min(A_{PQ},b_P+f_P-1,a_{PQ}+f_{QP}-1)\leq j<\min(b_P+f_P,a_{PQ}+f_{QP})$.

and

$C3_P$: **while** no termination **do**
begin do operation $S3_{PQ}$ or $R1_{PQ}$ for some neighbor Q;
if for all neighbors Q $a_{PQ}>b_P$
then begin $b_P:=b_P+1$; compute **end**
end.

Hence, compared to $S2_{PQ}$, sending in $S3_{PQ}$ is slightly less restricted for those links (P,Q) where $f_P>f_{QP}$. The system-wide invariants of the modified protocol skeleton now are as follows.

Lemma 4.1. Let processors P and Q share a link. Then

(i) A_{PQ}, a_{PQ} and b_P are nondecreasing.

(ii) $A_{PQ} \geq a_{PQ} - f_{PQ} \geq b_P - f_{PQ}$.

(iii) The number of possible values for j in $S3_{PQ}$ is at least 1 and at most $f_{PQ}+f_{QP}$.

(iv) $A_{PQ} \leq a_{QP}$.

(v) Let $M_{j_1}, \ldots, M_{j_r}$ be the contents of the P-to-Q queue from the front to the rear, where $r \geq 0$, and let j_{max} be the maximum index of any message that has ever been removed from the P-to-Q queue. (If nothing has ever been removed, let $j_{max}=-1$.) Let $j_0=j_{max}$ and $j_{r+1}=\min(b_P+f_P, a_{PQ}+f_{QP})-f_{PQ}-f_{QP}$. Then

$$j_k < j_{k'}+f_{PQ}+f_{QP} \text{ for } 0 \leq k < k' \leq r+1.$$

Proof. Note that $b_P \leq a_{PQ}$ from the specification of $C3_P$. The remaining proofs are analogous to those of the corresponding lemmas in [5]. Q.E.D.

Theorem 4.1. $b_Q - f_P \leq b_P \leq b_Q + f_Q$.

Proof. Compare the proof of theorem 3.1. Since j_{max} is again $A_{QP}+f_{QP}-1$, we have $A_{QP}+f_{QP}-1 < \min(b_P+f_P, a_{PQ}+f_{QP})$ by lemma 4.1.(v). Hence $b_Q \leq a_{QP} \leq A_{QP}+f_{QP} \leq \min(b_P+f_P, a_{PQ}+f_{QP}) \leq b_P+f_P$. Q.E.D.

Theorem 4.2. If M_j is in the P-to-Q queue, we have

$$b_Q - f_{QP} - f_{PQ} \leq a_{QP} - f_{QP} - f_{PQ} \leq j < \min(b_Q+f_Q+f_{QP}, a_{QP}+f_{QP}+f_{PQ}).$$

Proof. In the proof of theorem 4.1 we saw that $a_{QP} \leq \min(b_P+f_P, a_{PQ}+f_{QP})$. Hence $a_{PQ} \leq \min(b_Q+f_Q, a_{QP}+f_{PQ})$. From lemma 4.1.(v) we have $j < \min(b_P+f_P, a_{PQ}+f_{QP}) \leq a_{PQ}+f_{QP} < \min(b_Q+f_Q+f_{QP}, a_{QP}+f_{PQ}+f_{QP})$. Also $j \geq j_{max}+1-f_{PQ}-f_{QP} = A_{QP}+f_{QP}-f_{PQ}-f_{QP} \geq a_{QP}-f_{PQ}-f_{QP} \geq b_Q-f_{QP}-f_{PQ}$. Q.E.D.

Corollary 4.1. $b_P \leq a_{PQ} \leq b_P+f_{PQ}+f_P$.

Proof. From theorem 4.2 with "P" and "Q" interchanged. Q.E.D.

Hence in transmissions according to the new protocol skeleton it would be sufficient again to use sequence numbers modulo some n with $n \geq 2(f_{QP}+f_{PQ})$, but for the administration in every processor P of b_P and a_{PQ} for all neighbors Q we need at least $n \geq 1 + f_P + \max\{f_{PQ} \mid Q \text{ a neighbor of } P\}$.

Returning to the alternating bit protocol on all links (hence $f_{PQ}+f_{QP}=1$ for all links), we conclude that one bit suffices for transmission, but that we need two bits for the counter b_P. But this need not be so in the very special case that for each processor P and each neighbor Q of P, $f_P=f_{QP}$. In that case, operation S3 amounts to operation S2 because $a_P=b_P$ always, and hence one bit for b_P suffices. However, it can be shown that to be able to fulfill this condition, the network must be bipartite (because we need $f_{QP}=f_{RP}$ for all neighbors Q and R of P, for each processor P).

Using the alternating bit protocol means we have to fix the order of the send and receive operations. Since we want only one round of communication separated from the next round by a computation, we have to ensure that each link, after completing one round of communication, waits until the state counter is increased and it can begin its next round of communication. Otherwise, some links might try to reiterate the same round of communication over and over again. Hence we introduce a "wait for synchronization" statement of the form: **do** nothing **until** ... . If we define $f_P=1$ for all processors P to simplify the formulation, this gives us the following protocol.

The Alternating Bit Synchronizer:
Protocol for P's link to Q in case $f_{PQ}=0$.

$A_{PQ}:=-f_{PQ}$;
while no termination **do**
begin do nothing **until** $A_{PQ}+f_{PQ} \leq b_P$;
send message <M,A_{PQ} mod 2>;
receive message <N,bit>;
if bit=$(A_{PQ}+f_{PQ})$ mod 2
then begin store N; $A_{PQ}:=A_{PQ}+1$ **end**
end.

Protocol for P's link to Q in case $f_{PQ}=1$.

$A_{PQ}:=-f_{PQ}$;
while no termination **do**
begin do nothing **until** $A_{PQ}+f_{PQ}\leq b_P$;
receive message <N,bit>;
if bit=$(A_{PQ}+f_{PQ})$ mod 2
then begin store N; $A_{PQ}:=A_{PQ}+1$ **end**;
send message <M,A_{PQ} mod 2>
end.

Programcycle for processor P.

$b_P:=0$;
while no termination **do**
if for all neighbors Q $A_{PQ}+f_{PQ}>b_P$
then begin $b_P:=b_P+1$; compute **end.**

Note that the only difference in the formulation of the link protocols for $f_{PQ}=0$ or $f_{PQ}=1$, is in the order of the send and receive operations. It is easily seen that the alternating bit synchronizer is a more specific formulation of the protocol skeleton consisting of operations S3 and R1 together with programcycle C3, if we note that the relation $A_{PQ}=a_{PQ}-f_{PQ}$ is now a system-wide invariant. Hence we have local synchronization by theorem 4.1.

Theorem 4.3. One bit is sufficient for all variables in the alternating bit synchronizer.

Proof. Since $f_{PQ}+f_{QP}=1$ over all links (P,Q), we can use sequence numbers mod 2 in all messages. If $f_{PQ}=0$ we have $b_P\leq a_{PQ}=A_{PQ}\leq b_P+1$. If $f_{PQ}=1$ we have $b_P\leq a_{PQ}=A_{PQ}+1\leq b_P+2$. However, in this last protocol we introduced an extra waiting step to ensure synchronization. Hence we can improve the bound on a_{PQ} to $a_{PQ}\leq b_P+1$. In the beginning $a_{PQ}=A_{PQ}+1=0=b_P$. Upon receipt of a message either A_{PQ} stays the same or is increased by one. Hence $A_{PQ}=b_P$ or $A_{PQ}+1=b_P$. On the other hand, b_P can only be changed to b_P+1 if for all neighbors Q of P, $A_{PQ}+f_{PQ}>b_P$. In

particular this can only happen if $A_{PQ}+1>b_P$. Hence we had $A_{PQ}=b_P$, which changes to $A_{PQ}+1=b_P$ when b_P is increased by one. Moreover, the guard "do nothing until $A_{PQ}+1\leq b_P$" ensures that $A_{PQ}+1=b_P$ before we can do a next receive and hence possibly increase A_{PQ}. Thus $A_{PQ}=b_P$ or $A_{PQ}+1=b_P$ is invariant and we have $b_P\leq a_{PQ}=A_{PQ}+1\leq b_P+1$. Hence for all variables computing mod 2 suffices. Q.E.D.

Note that now it is not the case that if we know that $f_{PQ}=0$ and $f_{QP}=1$, and $b_P\neq b_Q$ that we can deduce $b_Q=b_P+1$. For example, we could have $b_Q=a_{QP}=a_{PQ}=b_P+1$ as well as $b_P=a_{PQ}=a_{QP}=b_Q+1$. Hence we do not have the problem with deadlock as in theorem 3.3. It does have the consequence that inspecting the bit values b_P and b_Q gives no information on which processor is behind which, but we do not need this anyway.

5. The sliding window synchronizer. The alternating bit protocol (and hence, the alternating bit synchronizer) has a basic asymmetry - one side must begin with sending and the other must wait with sending until it has received a correct message. This might be inconvenient in a network geared to true distributed computations. To deal with the asymmetry, we consider another modification of the basic synchronizer from section 3.

We could get around the asymmetry problem in the following way: allow all nodes to begin with sending, by taking $f_{PQ}=f_{QP}=1$ for all links (P,Q). However, this leads to a new problem. If we allow that messages are lost in a link, we now have no way to prevent that messages are received out of order. For example, P sends M_0, M_1, M_2 to Q and receives N_0, N_1 from Q. If M_1 is lost, Q does not send N_2 which would be an acknowledgement for M_1. Hence P times out and retransmits M_1. Thus Q receives M_0, M_2, M_1 and the original ordering is lost. Moreover, as a_{QP} and A_{QP} are increased, values are skipped. In this example a_{QP} attains the values 0,1,1,3 and A_{QP} the values -1,0,2,2 respectively. Hence we need a buffer for each link to store the messages that arrived "before their turn" (unless we throw them away and wait for a retransmission by P) and a buffer for the messages that still might have to be retransmitted. To be able to write down the protocol for

communication over a link we have to specify the waiting conditions for the beginning of a new "state" in terms of a_P,a_{PQ} and A_{PQ}. Hence we will first analyze the meaning of different combinations of values of these variables in the basic protocol of section 3. The programcycle remains unchanged at $C2_P$.

Lemma 5.1. Using programcycle C2 at all nodes and operations $S2_{PQ}$ and $R1_{PQ}$ with $f_{PQ}=f_{QP}=1$ for all links (P,Q) gives only four possible relations between a_P,a_{PQ} and A_{PQ}, namely

(i) $A_{PQ}=a_{PQ}-1=a_P-1$,

(ii) $A_{PQ}=a_{PQ}-1=a_P$,

(iii) $A_{PQ}=a_{PQ}-1=a_P+1$ and

(iv) $A_{PQ}=a_{PQ}+1=a_P+1$.

Proof. From lemma 3.1 and theorem 3.1 we have with $f_{PQ}=f_{QP}=1$ $a_P-1\leq a_{PQ}-1\leq A_{PQ}\leq a_P+1$. Hence $a_P-1=A_{PQ}$, or $a_P=A_{PQ}$ or $a_P+1=A_{PQ}$. If $a_P-1=A_{PQ}$ we know that $a_P=a_{PQ}$. The case $a_P=A_{PQ}$ implies $A_{PQ}=a_{PQ}$ or $A_{PQ}=a_{PQ}-1$, whereas $a_P+1=A_{PQ}$ gives rise to the three possibilities $A_{PQ}=a_{PQ}-1$, $A_{PQ}=a_{PQ}$ and $A_{PQ}=a_{PQ}+1$. However, consider the case that $A_{PQ}=a_{PQ}$ and recall the meanings of A_{PQ} and a_{PQ}, respectively. If A_{PQ} has value j, we know that message M_j has been received. If a_{PQ} has value j, we know that messages M_0 up to M_{j-1} have been received. But since M_j has been received too, a_{PQ} must at least have value j+1. Hence the case $A_{PQ}=a_{PQ}$ cannot occur and we are left with the four remaining relations (i) $A_{PQ}=a_{PQ}-1=a_P-1$, (ii) $A_{PQ}=a_{PQ}-1=a_P$, (iii) $A_{PQ}=a_{PQ}-1=a_P+1$ and (iv) $A_{PQ}=a_{PQ}+1=a_P+1$. Q.E.D.

What is the meaning of these four relations? Case (iv) $A_{PQ}=a_{PQ}+1=a_P+1$ corresponds with the situation that message $M_{A_{PQ}}$ is received from Q, while message $M_{A_{PQ}-1}=M_{a_{PQ}}$ is not, probably because it was lost. Hence, if we want to keep the synchronization, we can do nothing but wait for a retransmission of $M_{A_{PQ}-1}$. If it arrives, we end up in case (iii) $A_{PQ}=a_{PQ}-1=a_P+1$, since a_{PQ} will be increased by two and A_{PQ} and a_P have not changed. Thus we will have to wait until a_P is

increased and we have case (ii) $A_{PQ}=a_{PQ}-1=a_P$ to be allowed to do a new send operation corresponding to the next "state". Recall that the corresponding receive operation has already been done earlier. However, if up till now messages have been arriving in order, case (ii) $A_{PQ}=a_{PQ}-1=a_P$ corresponds to the situation where the communication over this link was finished and we are waiting for the transition to the next "state", after which we have case (i) $A_{PQ}=a_{PQ}-1=a_P-1$.

This analysis gives rise to the following protocol specification, apart from the time-out mechanism for retransmission of possible lost messages, which we omit.

The sliding window synchronizer.

Protocol for P's link to Q.

$a_{PQ}:=0$; $A_{PQ}:=-1$;
while no termination **do**
begin if $A_{PQ}=a_P+1$ **then**
begin while $A_{PQ}=a_{PQ}+1$ **do**
begin receive message N_i and optionally store it;
if $i>A_{PQ}$ **then** $A_{PQ}:=i$;
a_{PQ}:= new (maximum) value such that P has received and stored messages N_0, N_1, ..., $N_{a_{PQ}-1}$ from Q
end;
do nothing **until** $a_{PQ}-1=a_P$;
send message M_{a_P}
end else
begin do nothing **until** $a_P=a_{PQ}$;
send message M_{a_P};
receive message N_i and optionally store it;
if $i>A_{PQ}$ **then** $A_{PQ}:=i$;
a_{PQ}:= new (maximum) value such that P has received and stored messages N_0, N_1, ..., $N_{a_{PQ}-1}$ from Q
end
end.

The sliding window synchronizer is a more specific formulation of the protocol skeleton consisting of actions S2 and R1 together with programcycle C2, as is clear from the analysis above. Hence we have local synchronization by theorem 3.1. Of course in the case of the sliding window synchronizer two bits are sufficient for all variables. Hence we have to pay for getting rid of the asymmetry of the alternating bit protocol by an extra variable for each link (we now need both A_{PQ} and a_{PQ}), while we also need two bits in stead of one for all variables. The messages have become one bit longer too, and the protocol specification is more complicated.

6. Composition of sliding window protocols. Consider a chain of three processors P, Q and R, and the bidirectional links (P,Q) and (Q,R). Assume that the communication on the links is governed by sliding window protocols for some values of f_{PQ}, f_{QP}, f_{QR} and f_{RQ}. Assume further that Q's only task is to send each message it receives from P, on to R, and to send each message it receives from R, on to P. Thus Q acts as a relay between P and R. Is it possible to describe the communication between P and R by one sliding window protocol? It is clear that this asks for the compositionality of this sort of protocol.

Consider the basic protocol skeleton with actions S1 and R1 as defined in section 2. Recall that we defined

a_{PQ} = the number of consecutive messages that P has received from Q and stored.

A_{PQ} = the number of consecutive messages that P is sure Q has received and stored.

To be able to define the "hypothetical" variables a_{PR} and A_{PR} we should take $a_{PR}=a_{PQ}$ and $A_{PR}=A_{PQ}+f_{PQ}-f_{QR}$ to make the definitions meaningful. To see this, note that since Q sends only messages that it has received and stored from R, the number of consecutive messages that P has received from R is the same. Hence $a_{PR}=a_{PQ}$. Further, P knows A_{PQ} because it has received a message N_j from Q with $A_{PQ}= j-f_{PQ}+1$: an implicit acknowledgement from Q for message $M_{j-f_{PQ}}$ from P. However, we also know Q must have received N_j from R, which is an implicit acknowledgement for message $M_{j-f_{QR}}$ which Q sent to R. Thus Q sent $M_{j-f_{QR}}$ to R because it

received $M_{j-f_{QR}}$ from P. Hence now message N_j becomes an implicit acknowledgement that R received message $M_{j-f_{QR}}$ from P. Thus P should set A_{PR} to $j-f_{QR}+1$ and we should define $A_{PR}=A_{PQ}+f_{PQ}-f_{QR}$. From this discussion we can conclude that in order for P and R to communicate according to a sliding window protocol, the component protocols should have parameters $f_{PR}=f_{QR}$ and $f_{RP}=f_{QP}$!

Theorem 6.1. If P and Q communicate under a sliding window protocol with parameters f_{PQ} and f_{QP}, Q and R communicate under a sliding window protocol with parameters f_{QR} and f_{RQ}, and Q only sends messages to R and to P if it has received them from P and R respectively, then the communication between P and R satisfies the conditions of a sliding window protocol with parameters f_{QR} and f_{QP} if $f_{QR}\geq f_{PQ}$ and $f_{QP}\geq f_{RQ}$. If $f_{QR}=f_{PQ}$ and $f_{QP}=f_{RQ}$, the protocol descriptions for the P-to-R and R-to-P message transfers are exactly the same as those for the P-to-Q and R-to-Q message transfers.

Proof. Define $A_{PR}=A_{PQ}+f_{PQ}-f_{QR}$, $a_{PR}=a_{PQ}$, $f_{PR}=f_{QR}$, $A_{RP}=A_{RQ}+f_{RQ}-f_{QP}$, $a_{RP}=a_{RQ}$ and $f_{RP}=f_{QP}$. Let $f_{QR}\geq f_{PQ}$ and $f_{QP}\geq f_{RQ}$. We immediately have (see for example lemma 3.1): $A_{PR}=A_{PQ}+f_{PQ}-f_{QR}\geq a_{PQ}-f_{QR}=a_{PR}-f_{PR}$ (and $A_{RP}\geq a_{RP}-f_{RP}$). Consider the case that Q sends M_j to R. We know from the Q to R protocol that $\min(A_{QP},a_{QR}+f_{RQ}-1) \leq j < a_{QR}+f_{RQ}$. However, Q must have received M_j from P, too. Hence $j\leq j_{max}$ with j_{max} from the P-to-Q queue and thus $j\leq A_{QP}+f_{QP}-1$. This gives us $j<\min(a_{QR}+f_{RQ},A_{QP}+f_{QP})$. Now we can reformulate the system-wide invariant about the contents of the Q-to-R queue as follows:

Claim. Let $M_{j_1}, \ldots, M_{j_r}$ be the contents of the Q-to-R queue, from the front to the rear, where $r\geq 0$, and let j_{max} be the maximum index of any message that has ever been removed from the Q-to-R queue. (If nothing has ever been removed, let $j_{max}=-1$). Let $j_0=j_{max}$ and $j_{r+1}=\min(a_{QR}+f_{RQ},A_{QP}+f_{QP})-f_{RQ}-f_{QR}$, then

$$j_k < j_{k'}+f_{RQ}+f_{QR} \text{ for } 0\leq k<k'\leq r+1.$$

Proof. Initially the queue is empty, $j_0=-1$ and $j_{r+1}=-f_{RQ}-f_{QR}$, hence the relation $j_0 < j_{r+1}+f_{RQ}+f_{QR}$ holds. Operations $S1_{PQ}$, $R1_{PQ}$, $S1_{QP}$, $R1_{QP}$, $R1_{QR}$ and $S1_{RQ}$ do not change the contents of the Q-to-R queue. $R1_{QP}$ and $R1_{QR}$ might increase j_{r+1}, which preserves the inequalities too. $S1_{QR}$ adds an $M_{j_{r'}}$ at the end of the queue, with necessarily $\min(A_{QR},a_{QR}+frbq-1) \leq j_{r'} < \min(a_{QR}+f_{RQ},A_{QP}+f_{QP})$. Since $j_{r'} \geq a_{QR}-f_{QR}$, $j_k < j_{r+1}+f_{QR}+f_{RQ} = \min(a_{QR}+f_{RQ},A_{QP}+f_{QP}) \leq a_{QR}+f_{RQ} \leq j_{r'}+f_{QR}+f_{RQ}$. Also, $j_{r'} < \min(a_{QR}+f_{RQ},A_{QP}+f_{QP}) = j_{r+1}+f_{QR}+f_{RQ}$. Hence the invariant remains valid under $S1_{QR}$. $R1_{RQ}$ removes M_{j_1} from the front of the queue, and $j'_{max}=\max(j_{max},j_1)$. Since the invariant was valid for both j_{max} and j_1, it remains valid for the maximum of both. Q.E.D.

As a consequence, we have $j_{max} < j_{r+1}+f_{RQ}+f_{QR}$ and since $j_{max}=A_{RQ}+f_{RQ}-1$, $A_{RQ}+f_{RQ} \leq \min(a_{RQ}+f_{QR},A_{QP}+f_{QP}) \leq A_{QP}+f_{QP}$. The corresponding derivation on the Q-to-P queue gives us $A_{PQ}+f_{PQ} \leq A_{QR}+f_{QR}$. Thus $A_{PR} = A_{PQ}+f_{PQ}+f_{QR} \leq A_{QR}+f_{QR}-f_{QR} \leq a_{RQ}=a_{RP}$ (and $A_{RP} \leq a_{PR}$). Since $f_{PR}=f_{QR} \geq f_{PQ}$, $A_{PR} \leq A_{PQ}$ and since $f_{RP}=f_{QP}$ and $a_{PR}=a_{PQ}$ we have that $\min(A_{PQ},a_{PQ}-f_{QP}-1) \leq j < a_{PQ}+f_{QP}$ implies $\min(A_{PR},a_{PR}+f_{RP}-1) \leq j < a_{PR}+f_{RP}$. Since we know that for M_j in the Q-to-R queue we have (a consequence of the claim) $a_{RQ}-f_{RQ}-f_{QR} \leq j < a_{RQ}+f_{RQ}+f_{QR}$ and $f_{PR}+f_{RQ}=f_{QR}+f_{QP} \geq f_{QR}+f_{RQ}$, this implies that $a_{RP}-f_{RP}-f_{PR} \leq j < a_{RP}+f_{RP}+f_{PR}$. Hence the communication between P and R satisfies the conditions of a sliding window protocol with parameters $f_{PR}=f_{QR}$ and $f_{RP}=f_{QP}$, and the descriptions of $S1_{PQ}$, $R1_{PQ}$, $S1_{RQ}$ and $S1_{RQ}$ coincide with the descriptions of $S1_{PR}$, $R1_{PR}$, $S1_{RP}$ and $R1_{RP}$ respectively if $f_{QP}=f_{RQ}$ and $f_{QR}=f_{PQ}$. Q.E.D.

We leave it to the reader to check that the same theorem holds if P, Q and R use sending operation S2 instead of S1, and thus are synchronizing via their sliding window protocols.

Corollary 6.1. If $f_{QP} \geq f_{RQ}$ and $f_{QR} \geq f_{PQ}$ then $a_{RP}-f_{RP} = a_{RQ}-f_{QP} \leq a_{PQ} = a_{PR} \leq a_{RQ}+f_{QR} = a_{RP}+f_{PR}$. If S2 sending operations are used, then $a_R-f_{RP} \leq a_P \leq a_R+f_{PR}$.

Assume we have a network which is synchronized by the sliding

window synchronizer. Assume it contains a node Q with neighbors P and R, and Q only relays messages from P to R and vice versa. Then we can ignore Q and view the communication between P and R as if it is taking place over a direct link between P and R. Note that as a consequence of the relay character of Q, the synchronization between P and R has become closer too! Namely, we now have $a_R-1 \leq a_P \leq a_R+1$ instead of $a_R-2 \leq a_Q-1 \leq a_P \leq a_Q+1 \leq a_R+2$ which we had before.

7. Discussion. It is useful to compare the synchronizers we derived with the results of Awerbuch [1]. We saw that for the alternating bit synchronizer the synchronization follows more or less natural from the original alternating bit protocol. We note that it is similar to Awerbuch's synchronizer α [1]. However, apart from our weaker assumptions on the network model (we allow messages to be lost while Awerbuch does not), we use less communication overhead to achieve the synchronization. There is another difference in these synchronizers, and that is that synchronizer α ensures synchronization over the whole network, while the alternating bit synchronizer ensures only local synchronization between neighboring nodes. This is really all that is needed because, after all, only nodes that communicate directly over a link need to be synchronized, and the fact that nodes which do not communicate directly are not completely synchronized is usually irrelevant to the correctness of a parallel algorithm. Thus for synchronizer α the (virtual) state counters of any two processors in the network would differ at most one, while the alternating bit synchronizer can ensure only a difference of at most the diameter of the network. Awerbuch uses in each synchronization round i) the messages (to possibly only some) neighbors due to the simulated algorithm, ii) acknowledgements for these messages, and iii) "safe messages" to all neighbors. We use only messages to all neighbors, although their length is increased by one bit. If the simulated algorithm does not require messages to all neighbors, we can add dummy messages for synchronization. Thus it depends on the communication in the original algorithm whether the savings in communication are negligible or amount to a factor of three.

Awerbuch's objection to synchronizer α is its relatively high cost

of communication. For the alternating bit synchronizer the extra cost of communication is neglible in case the simulated algorithm uses a lot of communication (over all links), but quite substantial, though still less than for synchronizer α, in case the simulated algorithm uses only a small subset of links of the network for communication (e.g. only those that form a spanning tree for the network). However, there is a large class of algorithms where we can save the extra amount of communication due to the introduction of dummy messages for synchronization. These are the algorithms where the communication over one link has the following pattern: for some time the link is used regularly for communication, followed by a period where the link is never used again. Think for example of algorithms that begin by building up some kind of spanning tree, and that later confine all communication to the links of that spanning tree. What we can do in that case is using only the links that are currently active in communication for synchronization. This would mean substituting "all currently active links (P,Q)" for "all neighbors Q" in programcycle C2 of processor P. Unfortunately, though it is easy to declare a link "dead" for synchronization purposes, it is in general not possible to bring it to life again, since the invariants of the protocol prescribe that neighbors have state counters that differ by at most one. Hence it is not possible to choose correct values for the protocol variables to begin communication over a link if the state counters of the nodes differ by more than one, and we only know that they differ at most the number of active links between them. Note however that we do not encounter this difficulty in starting up the alternating bit synchronizer, since a node can only leave its initial state of zero after it has woken up all its neighbors and has received messages from them.

References.

[1] Awerbuch, B., Complexity of Network Synchronization, J. ACM 32(1985) 804-823.

[2] Knuth, D.E., Verification of link-level protocols, BIT 21(1981) 31-36.

[3] Krogdahl, S., Verification of a class of link-level protocols, BIT 18(1978) 436-448.

[4] Schoone, A.A. and J. van Leeuwen, An alternate view of the alternating bit protocol, in: H.Noltemeier (Ed.), Proc. WG'85 (Int. Workshop on Graph-theoretic Concepts in Computer Science), Trauner Verlag, Linz, 1985, pp. 347-354.

[5] Schoone, A.A. and J. van Leeuwen, Verification of balanced link-level protocols, Techn. Rep. RUU-CS-85-12, University of Utrecht, Utrecht, 1985. (Submitted for publication.)