LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-426

NON-OBTRUSIVE SYNCHRONIZERS

Baruch Awerbuch
David Peleg

April 1990

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Non-Obtrusive Synchronizers

Baruch Awerbuch * David Peleg †

April 2, 1990

Abstract

The synchronizer is a simulation methodology for simulating a synchronous network by an asynchronous one, thus enabling the execution of a synchronous algorithm on an asynchronous network. Previously known synchronizers are obtrusive, in the sense that they require each processor in the entire network to participate in the synchronization process, even if the protocol executed involves only a small sub-network, and the rest of the processors would much prefer being left alone. Obtrusiveness can be evaluated by means of the communication complexity of the synchronizer: a non-obtrusive synchronizer is one whose communication complexity depends only on the number of processors active in the synchronized protocol, and not on the total number of processors in the network. This paper presents a non-obtrusive synchronizer, whose complexities are at most a $polylog(n)$ factor away from the optimum.

*Dept. of Mathematics and Lab. for Computer Science, M.I.T., Cambridge, MA 02139; Supported by Air Force Contract TNDGAFOSR-86-0078, ARO contract DAAL03-86-K-0171, NSF contract CCR8611442, and a special grant from IBM.

†Department of Applied Mathematics, The Weizmann Institute, Rehovot 76100, Israel. Supported in part by an Allon Fellowship and by a Bantrell Career Development Fellowship.

1 Introduction

The synchronizer is a simulation methodology for simulating a synchronous network by an asynchronous one, thus enabling the execution of a synchronous algorithm on an asynchronous network. The availability of such a tool enables us to design distributed network algorithms under the assumption of a synchronous environment, which makes it easier to comprehend and reason about the behavior of the algorithm. Synchronizers were defined in [Awe85a], and various aspects or applications of them were further studied in [Awe85b, CGZ86, SM86, FLS87, LT87, PU89, AS88, ER90].

In many cases, the protocol that needs to be executed involves only a small sub-network, and the rest of the processors in the network are not required by the problem specification to participate in it. In such a case, it would be highly desirable to allow these processors to stay out of the execution in the synchronized process (i.e., on the asynchronous network) as well. Unfortunately, previously known synchronizers have the undesirable property (henceforth referred to as *obtrusiveness*) that they require each processor in the entire network to participate in the synchronization process. In fact, each processor is required to send some messages in every pulse of the run.

Obtrusiveness can be evaluated by means of the communication complexity of the synchronizer. An obtrusive synchronizer is one whose communication complexity depends on the total number of processors in the network, n. In contrast, a non-obtrusive synchronizer is one whose communication complexity depends only on the number of processors active in the protocol π, denoted v_π.

The synchronizer operates by generating a sequence of local clock pulses at each processor of the network, satisfying the following property:

> pulse number p is generated by a processor only after it received all the messages of the algorithm sent to it by its neighbors during their pulse number $p-1$.

Intuitively, the problem lies in the fact that in case processor v did not send any message to its neighbor u at a certain pulse, u cannot deduce this by simply waiting for a fixed period of time, as link delays in the asynchronous network are unpredictable. The solution proposed in [Awe85a] is based on first requiring every processor receiving a message from a neighbor to send back an acknowledgment. This enables every processor to learn, within finite time, that all the messages it sent during a particular pulse have arrived. Such a processor is said to be *safe* with respect to that pulse. Note that the acknowledgments do not increase the message complexity or the time complexity of the algorithm by more than a constant factor.

A processor may generate a new pulse whenever it learns that all its neighbors are safe with respect to the current pulse. Thus the second and main phase of the synchronizer involves delivering this "safety" information. This phase is thus responsible for the additional time and message requirements per pulse, denoted by C_{pulse} and T_{pulse}.

The complexities of a synchronous algorithm S are related to those of the asynchronous algorithm A resulting from combining S with a synchronizer ν by $C_A = C_S + T_S \cdot C_{pulse}$ and $T_A = T_S \cdot T_{pulse}$. (We ignore, for the purposes of the present discussion, any additional costs of an initialization phase which may be needed for setting the synchronizer up.)

The complexities of the synchronizers α, β, γ of [Awe85a] and δ of [PU89] all involve $C_{pulse} = \Omega(n)$. Thus at every pulse, essentially every processor must participate in the collective effort of synchronizing the network.

This paper presents a non-obtrusive synchronizer. Its complexities are

$$\begin{aligned} C_{pulse} &= O(v_\pi \cdot \log^3 n) \\ T_{pulse} &= O(\log^2 n) \end{aligned}$$

These complexities are at most a $polylog(n)$ factor away from the optimum.

2 The model

We consider the standard model of an asynchronous point-to-point communication network (e.g. [GHS83]). The network is described by an undirected graph $G = (V, E)$. The nodes of the graph represent the processors of the network and the edges represent bidirectional communication channels between the processors. We use V (respectively, E) also to denote the *number* of vertices (resp., edges). The diameter of the network is denoted by $D(G)$, and we denote $\delta = \lceil \log_2 D(G) \rceil$.

All the processors have distinct identities. There is no common memory, and algorithms are event-driven (i.e., processors cannot access a global clock in order to decide on their action). Messages sent from a processor to its neighbor arrive within some finite but unpredictable time. Each message contains a fixed number of bits, and therefore carries only a bounded amount of information.

A synchronous network is a variation of the above model in which all link delays are bounded. More precisely, each processor keeps a local clock, whose pulses must satisfy the following property. A message sent from a processor v to its neighbor u at pulse p of v must arrive at u before pulse $p+1$ is generated by u.

Our complexity measures are defined as follows. The *communication complexity* of an algorithm A, C_A, is the worst-case total number of messages sent during the run of the algorithm. The *time complexity* of an algorithm A, T_A, is defined as follows. For a synchronous algorithm, T_A is the number of pulses generated during the run. For an asynchronous algorithm, T_A is the worst-case number of time units from the start of the run to its completion, assuming that each message incurs a delay of at most one time unit. This assumption is used only for performance evaluation, and does not imply that there is a bound on delay in asynchronous networks.

3 Preliminaries

3.1 Basic properties of local synchronizers

The main new ingredient for the local case is that nodes may be sleeping, and in that case we would hate to have to wake them up just in order to synchronize.

Notation 3.1 It is assumed that if a processor v is awaken by a message carrying a pulse number p, it sends its first messages only at pulse $p+1$. We let $first(v) = p+1$, the first pulse in which v sends any messages.

We denote by $\texttt{Nodes}(p)$ the set of all nodes that participate in the protocol by pulse p. Specifically, if $first(v) = p+1$ then $v \in \texttt{Nodes}(p+1) - \texttt{Nodes}(p)$. Also, denote

$$\texttt{Nodes}_i(v,p) = \texttt{Nodes}(p) \cap \mathcal{N}_{2^i}(v),$$

i.e., the set of nodes at distance 2^i from v that participate in pulse p.

We state the following trivial facts for future use.

Fact 3.2 For every vertex v,

1. if $i' \leq i$ and $p' \leq p$, then $\texttt{Nodes}_{i'}(v,p') \subseteq \texttt{Nodes}_i(v,p)$.
2. $\texttt{Nodes}_i(v,p) = \texttt{Nodes}_\delta(v,p)$ for every $i \geq \delta$.

Accordingly, we require a *generalized safety principle*, whose definition is stated as follows:

Definition 3.3 The processor v is said to be *safe* with respect to pulse p, denoted $\textsc{Safe}(v,p)$, if either $v \notin \texttt{Nodes}(p)$, or $v \in \texttt{Nodes}(p)$ and all the messages it sent during pulse p have arrived.

In order to detect safety, for every message in the protocol, an acknowledgement is sent. In particular, ack indicates whether the sender has been chosen as a parent or not. Once all acks have been collected for messages of a particular pulse, node is declared to be safe w.r.t. this pulse.

The following trivial fact will be used later without reference.

Fact 3.4 If v satisfies $\textsc{Safe}(v,p)$ then also $\textsc{Safe}(v,q)$ for every $q < p$.

As in the usual, global case, we characterize a processor's readiness to increase its pulse in terms of its neighbors' safety.

Definition 3.5 the processor w is *done* with pulse p (i.e., it is ready to increase its counter to pulse $p+1$), denoted $\textsc{Done}(w,p)$, if all the messages sent to w from its neighbors at pulse p have already arrived.

Fact 3.6 A processor w satisfies $\textsc{Done}(w,p)$ if each of its neighbors v satisfies $\textsc{Safe}(v,p)$. ∎

3.2 Execution trees

When a processor v is awaken by a message for the first time, it remembers the sender w of the message as its *parent*, and sets $parent(v) = w$. (for originators v, $parent(v) = nil$). The *parent* relation defines a directed subgraph called the *execution forest*, with one tree per originator. Let *child* be the inverse relation, i.e., $u \in child(v)$ iff $v = parent(u)$.

In order to control processors locally, we would like to keep track of the lowest (most recent) layer of the execution tree. The depth of this layer depends on p as follows. Throughout the paper we associate with every pulse number p a *level* number $\ell(p)$, defined as follows. Let $i \geq 0$, $j \geq 1$ be integers such that $p = (2j-1) \cdot 2^i$. Then $\ell(p) = \min\{i, \delta\}$. Hence there are $\delta = \lceil \log_2 D(G) \rceil$ possible levels.

We associate with the pulse p a forest, denoted $tree_p$, as follows. Let $i = \ell(p)$. The set of *child nodes* of $tree_p$ is

$$\texttt{Children}_p = \{u \mid p - 2^i < first(u) \leq p\}.$$

The set of *roots* of $tree_p$ is

$$\texttt{Roots}_p = \{v \mid first(v) \leq p - 2^i,\ \exists u \in \texttt{Children}_p\ (v = parent(u))\}.$$

The forest $tree_p$ is then the subgraph of the execution tree induced by $\texttt{Children}_p \cup \texttt{Roots}_p$. Thus $tree_p$ is a forest of subtrees of the execution forest, all of depth 2^i, consisting of all the trees forming the layer of depth $p - 2^i + 1$ to p in the execution forest.

For every vertex $v \in tree_p$, let $tree_p(v)$ denote the (connected) tree containing v in $tree_p$, and let $root_p(v)$ denote the root of $tree_p(v)$.

Observe that when node u wakes up it knows $parent(u)$, and it is straightforward to ensure that the parent v knows that u enters $child(v)$ at this pulse.

3.3 The concept of a regional matching

The basic components of our construction are a read set $\texttt{Read}(v) \subseteq V$ and a write set $\texttt{Write}(v) \subseteq V$, defined for every vertex v. Consider the collection $\mathcal{RW}$ of all pairs of sets, namely

$$\mathcal{RW} \;=\; \{\, \texttt{Read}(v),\ \texttt{Write}(v) \mid v \in V \,\}.$$

Definition 3.7 The collection $\mathcal{RW}$ is an *m-regional matching* (for some integer $m \geq 1$) if for all $v, u \in V$ such that $dist(u,v) \leq m$, $\texttt{Write}(v) \bigcap \texttt{Read}(u) \neq \emptyset$.

4 The construction of the local synchronizer

For every level $0 \leq i \leq \delta$, and every node v, we define "read" and "write" sets, denoted by $\texttt{Read}_i(v)$ and $\texttt{Write}_i(v)$. Every node w in every set $\texttt{Write}_{i+1}(v)$ maintains lists $\texttt{List}_i^w(p)$ for every pulse p. These lists keep track of processors marked as *clean* w.r.t. pulse p. This notion is defined as follows.

Definition 4.1

- A node v is said to satisfy $\textsc{Listed}_i(v,p)$ if it is marked in each of the lists $\texttt{List}_j^w(p)$, for every $0 \leq j \leq i$ and every node $w \in \texttt{Write}_{j+1}(v)$.
- A processor v may mark itself in any of the lists $\texttt{List}_j^w(p)$ only when it is *clean* with respect to pulse p, denoted $\textsc{Clean}(v,p)$
- A node $v \notin \texttt{Roots}_p$ satisfies $\textsc{Clean}(v,p)$ iff it satisfies $\textsc{Safe}(v,p)$.
- A node $v \in \texttt{Roots}_p$ satisfies $\textsc{Clean}(v,p)$ iff each of its descendants $x \in tree_p(v)$ (excluding itself) satisfies $\textsc{Listed}_{\ell(p)}(x,p)$.

During the course of attempting to increase its pulse number from p to $p+1$, the node v invokes two procedures, $\textsc{Mark\_List}^v(p)$ and $\textsc{Scan}^v(p)$. Let us next specify the roles of these procedures.

Specification of $\textsc{Mark\_list}^v(p)$**:**

- *Precondition:* $\textsc{Clean}(v,p)$.
- *Effect:* v adds itself to all the lists $\texttt{List}_j^w(p)$ at all nodes $w \in \texttt{Write}_{j+1}(v)$, for every $0 \leq j \leq \ell(p)$. Thus upon termination of this procedure, $\textsc{Listed}_{\ell(p)}(v,p)$ holds.

Specification of $\textsc{Scan}^v(p)$**:**

- *Precondition:* Pulse p has been reached, v satisfies $\textsc{Listed}_{\ell(p)}(v,p)$, and v wishes to increase its pulse number to $p+1$.
- *Effect:* v invokes in parallel, at all $w \in \texttt{Read}_{\ell(p)+1}(v)$, the sub-procedure $\textsc{Verify}^w(v,p)$ at w, which returns back to v at the first time that

$$\texttt{List}^w_{\ell(p)}(p-2^j) \subseteq \texttt{List}^w_{\ell(p)}(p).$$

The node algorithm is thus as follows. *Before* raising pulse from p to $p+1$, node v has to mark itself, using procedure $\textsc{Mark\_list}^v(p)$, and complete all $\textsc{Scan}^v(p)$ operations. Schematically, the order of operations is described in Fig. 1.

A processor v that is awaken by a message carrying pulse number p needs to send an acknowledgement back to the sender, view itself as satisfying $\textsc{Safe}(v,p)$ and proceed to execute the algorithm of Fig. 1 in order to increase its pulse to $p+1$.

5 Correctness

Proving the correctness of the proposed synchronizer is centered around the following lemma.

Lemma 5.1 Consider a pulse p, and let $i = \ell(p)$ and $\hat{p} = p - 2^i$. Then,

1. for all v and for all $x \in \texttt{Nodes}_{i+1}(v,\hat{p})$, x satisfies $\textsc{Listed}_i(x,\hat{p})$ upon *invocation* of $\textsc{Scan}^v(p)$.
2. for all v and for all $x \in \texttt{Nodes}_{i+1}(v,\hat{p})$, x satisfies $\textsc{Clean}(x,p)$ upon *termination* of $\textsc{Scan}^v(p)$.
3. for all v and for all $x \in \texttt{Nodes}_i(v,p)$, x satisfies $\textsc{Listed}_i(x,p)$ upon *termination* of $\textsc{Scan}^v(p)$.

1. When node v reachs pulse p:
2. Send pulse p messages; receive acknowledgements
3. once $\textsc{Safe}(v,p)$:
4. **Case:**
   (a) $v \notin \mathit{tree}_p$: invoke $\textsc{Mark\_list}^v(p)$
   (b) $v \in \texttt{Children}_p$:
      i. invoke $\textsc{Mark\_list}^v(p)$
      ii. once v and all $u \in \mathit{child}(v)$ (if exist) reported termination of $\textsc{Mark\_list}^u(p)$: report termination of $\textsc{Mark\_list}^u(p)$ to $\mathit{parent}(v)$
   (c) $v \in \texttt{Roots}_p$: once all $u \in \mathit{child}(v)$ reported termination of $\textsc{Mark\_list}^u(p)$, call $\textsc{Mark\_list}^v(p)$.
5. once $\textsc{Mark\_list}^v(p)$ terminates, invoke $\textsc{Scan}^v(p)$ and await its termination.
6. Now, a new pulse $p+1$ may be started.

Figure 1: Algorithm for node v.

Proof: We proceed by induction on p. We assume that all the claims are valid for $p' < p$ and show that they must hold for p as well. The basis for the induction is the fact that the claims trivially hold for $p = 0$. Let $\hat{i} = \ell(\hat{p})$. Note that $\hat{i} \geq i$, with equality holding only if $\hat{i} = i = \delta$.

Proof of Property (1): Consider a node $x \in \texttt{Nodes}_{i+1}(v, \hat{p})$. Since either $\hat{i} \geq i+1$ or $\hat{i} = i = \delta$, by Fact 3.2, $x \in \texttt{Nodes}_{\hat{i}}(v, \hat{p})$. By induction hypothesis on Property (3), applied to $v, \hat{p}, \hat{i}$, we get that x satisfies $\textsc{Listed}_{\hat{i}}(x, \hat{p})$ upon termination of $\textsc{Scan}^v(\hat{p})$. Since $\hat{i} \geq i$, x certainly satisfies $\textsc{Listed}_i(x, \hat{p})$ at that time. Also, since $\hat{p} < p$, $\textsc{Scan}^v(\hat{p})$ *terminates* before $\textsc{Scan}^v(p)$ is *invoked.* The claim follows. •

Proof of Property (2): Consider $x \in \texttt{Nodes}_{i+1}(v, \hat{p})$. By Property (1), x satisfies $\textsc{Listed}_i(x, \hat{p})$ upon invocation of $\textsc{Scan}^v(p)$. Therefore $x \in \texttt{List}_i^w(\hat{p})$ for every $w \in \texttt{Write}_{i+1}(x)$ at that time. At this point, we need to make use of the fact that $\texttt{Write}_{i+1}(x) \cap \texttt{Read}_{i+1}(v) \neq \emptyset$, and thus there exists some $w \in \texttt{Read}_{i+1}(v)$ such that $x \in \texttt{List}_i^w(\hat{p})$ upon invocation of $\textsc{Scan}^v(p)$. This property must hold also upon invocation of $\textsc{Verify}^w(v, p)$ at w, as the former time clearly precedes the latter. ¿From the specification of $\textsc{Verify}^w(v, p)$, upon its termination $\texttt{List}_i^w(\hat{p}) \subseteq \texttt{List}_i^w(p)$, hence $x \in \texttt{List}_i^w(p)$. But x invokes $\textsc{Mark\_List}^x(p)$ and gets marked in the list $\texttt{List}_i^w(p)$ only after it satisfies $\textsc{Clean}(x, p)$. The claim follows. •

Proof of Property (3): For an $x \in \texttt{Nodes}_i(v, p)$, select r_x as

$$r_x = \begin{cases} x, & \text{if } x \in \texttt{Nodes}_i(v, \hat{p}) \\ root_p(x), & \text{otherwise} \end{cases}$$

We now argue that $r_x \in \texttt{Nodes}_{i+1}(v, \hat{p})$. This is immediate if $r_x = x$. Otherwise, the fact that $r_x = root_p(x)$ has the following two implications. First, $dist(r_x, x) \leq 2^i$, and hence the fact that $x \in \mathcal{N}_i(v)$ implies $r_x \in \mathcal{N}_{i+1}(v)$. Secondly, $first(r_x) \leq p - 2^i = \hat{p}$, so $r_x \in \texttt{Nodes}(\hat{p})$. Put together, indeed $r_x \in \texttt{Nodes}_{i+1}(v, \hat{p})$.

Applying Property (2) implies that, upon termination of $\textsc{Scan}^v(p)$, r_x satisfies $\textsc{Clean}(r_x, p)$. Since $x \in tree_p(r_x)$, it follows that x satisfies $\textsc{Listed}_i(x, p)$ at that time. The claim follows. •

This completes the proof. ∎

Corollary 5.2 A node v enters pulse $p + 1$ only when it satisfies $\textsc{Done}(v, p)$.

Proof: Processor v increases its pulse number only upon termination of $\textsc{Scan}^v(p)$. By Property (3) of the Lemma, at that time every node $x \in \texttt{Nodes}_i(v, \hat{p})$ satisfies $\textsc{Listed}_i(x, p)$, for $i = \ell(p)$. In particular, this holds for all neighbors x of v s.t. $x \in \texttt{Nodes}(p)$, hence every neighbor w of v already satisfies $\textsc{Safe}(w, p)$. ∎

6 Complexity analysis

For any m-regional matching $\mathcal{RW}$ define the following parameters.

$$\begin{aligned}
Deg_{write}(\mathcal{RW}) &= \max_{v \in V} |\mathtt{Write}(v)| \\
Rad_{write}(\mathcal{RW}) &= \frac{1}{m} \cdot \max_{u,v \in V} \{dist(u,v) \mid u \in \mathtt{Write}(v)\} \\
Deg_{read}(\mathcal{RW}) &= \max_{v \in V} |\mathtt{Read}(v)| \\
Rad_{read}(\mathcal{RW}) &= \frac{1}{m} \cdot \max_{u,v \in V} \{dist(u,v) \mid u \in \mathtt{Read}(v)\}
\end{aligned}$$

Now, observing the protocol of local synchronizer, we state the following key observation.

Theorem 6.1 The total communication and time overhead per pulse are

$$\begin{aligned}
C_{pulse} &= v_\pi \cdot \delta \cdot (Rad_{read} + Rad_{write}) \cdot (Deg_{read} + Deg_{write}) \\
T_{pulse} &= \delta \cdot (Rad_{read} + Rad_{write})
\end{aligned}$$

Proof: Whenever a node v reaches pulse p, it makes a constant numbers of accesses to the vertices of the sets $\mathtt{Write}_{i+1}(v)$ and $\mathtt{Read}_{i+1}(v)$, for $i = \ell(p)$, in the procedures MARK_LIST$^v(p)$ and SCAN$^v(p)$.

Suppose that all those procedures have been called at time t_0. Now, consider time $t_1 = t_0 + 2^{i+1} \cdot Rad_{write}$. By time t_1, all $\mathtt{Safe\_list}_i^w(p)$ lists are completed. Now, consider time $t_2 = t_1 + 2 \cdot 2^{i+1} \cdot Rad_{write}$. At this time, all MARK_LIST$^v(p)$ procedures must terminate. From this time t_2 and on, the process of convergence on the tree takes only 2^i time, i.e., it is proportional to the height of the tree. Also, observe that the $\mathtt{List}_i^w(p)$ lists are fully updated by time $t_3 = t_2 + 2^i + 2^{i+1} \cdot Rad_{write}$. From t_3 and on, SCAN$^v(p)$ procedures only need $2^i Rad_{read}$ time, i.e., proportional to the radius of read-sets. It follows that all the procedures terminate by time $t_4 = t_0 + O(2^i \cdot (Rad_{read} + Rad_{write}))$.

This time investment is amortized over 2^i time, yielding a charge of $O(Rad_{read} + Rad_{write})$ per pulse per level. The total charge per pulse becomes $O(\delta \cdot (Rad_{read} + Rad_{write}))$ per pulse, as every pulse may be charged by at most δ levels.

The account for communication is similar; the communication per node exceeds the time

per pulse by an $O(Deg_{read} + Deg_{write}))$ factor. The total charge per node per pulse in communication becomes $O(\delta \cdot (Rad_{read} + Rad_{write}) \cdot (Deg_{read} + Deg_{write}))$. ∎

We now make use of the following lemma of [AP89].

Lemma 6.2 [AP89] For all $m, k \geq 1$, it is possible to construct an m-regional matching $\mathcal{RW}_{m,k}$ with

$$\begin{aligned} Deg_{read}(\mathcal{RW}_{m,k}) &\leq \log n \\ Rad_{read}(\mathcal{RW}_{m,k}) &\leq 2\log n + 1 \\ Deg_{write}(\mathcal{RW}_{m,k}) &\leq 4 \\ Rad_{write}(\mathcal{RW}_{m,k}) &\leq 2 \cdot \log n + 1 \end{aligned}$$

Combining the lemma with Thm. 6.1, we deduce

Corollary 6.3

$$\begin{aligned} C_{pulse} &= O(v_\pi \cdot \delta \cdot \log^2 n) \\ T_{pulse} &= O(\delta \cdot \log n) \end{aligned}$$

References

[AP89] Baruch Awerbuch and David Peleg. Online tracking of mobile users. Unpublished manuscript, August 1989.

[AS88] Baruch Awerbuch and Michael Sipser. Dynamic networks are as fast as static networks. In 29^{th} *Annual Symposium on Foundations of Computer Science*, pages 206–220. IEEE, October 1988.

[Awe85a] Baruch Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, October 1985.

[Awe85b] Baruch Awerbuch. Reducing complexities of the distributed max-flow and breadth-first-search algorithms by means of network synchronization. *Networks*, 15(4):425–437, Winter 1985.

[CGZ86] C.T. Chou, I.S. Gopal, and S. Zaks. Synchronizing asynchronous bounded-delay networks. Research Report RC-12274, IBM Yorktown, October 1986.

[ER90] Shimon Even and Sergio Rijsbaum. The use of a synchronizer yields maximum computation rate in distributed networks. In *Proceedings of the* 22^{st} *Annual ACM Symposium on Theory of Computing, Baltimore, Maryland.* ACM SIGACT, ACM, May 1990.

[FLS87] A. Fekete, N. Lynch, and L. Shrira. A modular proof of correctness for a network synchronizer. In *Proceedings of the Amsterdam Workshop on Distributed Algorithms.* CWI, July 1987.

[GHS83] Robert G. Gallager, Pierre A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Prog. Lang. and Syst.*, 5(1):66–77, January 1983.

[LT87] K.B. Lakshmanan and K. Thulasiraman. On the use of synchronizers for asynchronous communication networks. In *Proceedings of the Amsterdam Workshop on Distributed Algorithms.* CWI, July 1987.

[PU89] David Peleg and Jeffrey D. Ullman. An optimal synchronizer for the hypercube. *SIAM J. Comput.*, 18(2):740–747, 1989.

[SM86] Baruch Schieber and Shlomo Moran. Slowing sequential algorithms for obtaining fast distributed and parallel algorithms: Maximum matchings. In *Proceedings of the* 5^{th} *Annual ACM Symposium on Principles of Distributed Computing*, pages 282–292. ACM, August 1986.