Tentative and definite distributed computations: an optimistic approach to network synchronization*

J. Garofalakis, P. Spirakis, B. Tampakas

Computer Technology Institute and Computer Science and Engineering Department, Patras University, Kolokotroni 3, 26110 Patras, Greece

S. Rajsbaum

Instituto de Matemáticas U.N.A.M., Mexico City, Mexico

Abstract

Garofalakis, A., P. Spirakis, B. Tampakas and S. Rajsbaum, Tentative and definite distributed computations: an optimistic approach to network synchronization, Theoretical Computer Science 128 (1994) 63–74.

We present here a general and efficient strategy for simulating a synchronous network by a network of limited asynchrony. Our proposed synchronizer is *optimistic* in the sense that it uses very efficient but *tentative protocols* to simulate a contiguous block of synchronous steps. However, since a tentative execution does not guarantee correct simulation, we *audit* the computation at selected points. The audits are used to check whether the computation of the block can be *certified* to be correct. We show that a wide class of networks of limited asynchrony admits practical tentative protocols which are highly likely to produce a correct simulation of one step with very small overhead. For those networks, the synchronizer exhibits a trade off between its communication and time complexities which is below the lower bounds for deterministic synchronizers. On one extreme the amortized complexity of our synchronizer is $O(1)$ messages and $O(\log n)$ time (expected) per "step" of the simulated synchronous protocol. On the other extreme the communication complexity is $O(e/\Delta^2)$ and the time complexity is $O(\log \Delta)$, for networks with e edges and maximum degree Δ.

Correspondence to: P. Spirakis, Computer Technology Institute and Computer Science and Engineering Department, Patras University, Kolokotroni 3, 26110 Patras, Greece.

* This research has been partially supported by the EEC ESPRIT Basic Research Action project No 7141 (ALCOM 2). A previous version of this paper appeared in the International Workshop on Distributed Algorithms (WDAG 92).

1. Introduction

1.1. *The need for synchronizers*

Consider a network of n processors and maximum degree Δ. The processors communicate by sending messages along e communication channels (edges). Assume that a program has been written for a *synchronous* network operation: On a global start-up signal, all processors start computing simultaneously. On every beat of the global clock each processor, according to its program, performs one computational step and sends messages to some of its neighbors. The transmission delay in the communication channels guarantees here that all messages arrive at their destinations in time to be used in the next computational step. We do not care about the exact nature and purpose of the program and we assume that processors and channels are reliable.

If one wants to run the same program on an *asynchronous network* of the same topology, where no global start-up signal exists and where transmission delays are unpredictable, then certain measures have to be taken to keep the computation correct. The use of *synchronizers* was suggested by Awerbuch [1] in order to simulate synchronous networks by asynchronous ones.

1.2. *Complexity measures*

The *communication complexity* of a synchronous algorithm π, $\mathscr{C}(\pi)$, is the worst-case number of messages sent during the run of the algorithm. The *time complexity* of a synchronous algorithm π, $\mathscr{T}(\pi)$, is the number of beats generated during the run of the algorithm. In our paper, for an asynchronous algorithm, the communication complexity is the worst case (among all possible starting time paterns) *expected* number of messages sent during a run, and the time complexity is the expected time of execution of a run. The expectation is taken with respect to a class of distributions of the transmission delays.

We are interested in studying what are the complexities of executing a synchronous program π on an asynchronous network of limited asynchrony. Namely, the complexities of a synchronizer S in our paper, are as follows: The time complexity is the worst case (among all synchronous programs π) expected duration of the execution of a (synchronous) step of π by the synchronizer S. The message complexity of S is the expected message overhead per step added by the synchronizer.

1.3. *Previous work*

For networks of unlimited asynchrony, Awerbuch presented synchronizers whose communication – time tradeoff is proved to be within a constant factor of the lower bound. The problem of designing efficient synchronizers has been studied in the past [2, 8]. Even and Rajsbaum examined the performance of synchronizer-controlled networks which have a global clock but no global start-up signal and whose transmission delays are either negligible [5] or fixed [6]. The results were generalized to other protocols by Malka and Rajsbaum. The performance of the synchronizer of [5, 6] under random transmission delays and processing times was analyzed in [9]. It was shown that any synchronizer has time delay (average) per step of $\Omega(\log \Delta)$.

Most of the above techniques provide *per-step synchronization*: the execution of the next "synchronous" step (of the original ideal synchronous program) begins only after the current "synchronous" step is *guaranteed* to have finished correctly. In such synchronizers, a communication (number of messages per "synchronous step" simulation) penalty of at least $\Theta(n)$ is paid, where n is the number of network nodes.

An exception to the above is the synchronizer alpha of [1]. In this synchronizer, the network essentially runs free, locally delaying the computation only as long "as necessary". In particular, each processor waits for messages to arrive from all its neighbors before it performs the next computation step (it is assumed that every message is followed by an "end-of-message" marker, even if the message is empty). A similar mechanism was used by Chandy and Lamport [4] and the whole approach is also encountered in models of marked graphs (see e.g. [3]).

1.4. Our results

To avoid the possibly long waits introduced by the simple synchronizer presented above, we propose that each processor waits only for a certain amount W of steps, hoping that all messages from neighbors will indeed arrive with high likelihood. This *optimistic* approach provides *tentative* executions which do not guarantee the correctness of the computation. Thus, we use a *definite* synchronization scheme, only at certain selected points, to *audit* the network's computations. The definite protocol checks whether the whole sequence of many tentative simulated steps can be *certified* to be correct. If not, the network's computation is *rolled back* to the previous audit point, and we restart the computation from there. A similar scheme was also employed in [7] to provide robust parallel computations on faulty parallel random access machines (PRAMs) (see [11]). Our work shows how to apply such ideas to get efficient synchronizers. For a wide class of networks we show our optimistic synchronizer to get an amortized mean delay of $O(\beta \log m)$ per "synchronous" step and an amortized mean number of messages of $O(e/m^{\theta})$ per "synchronous" step, for a parameter m ($\Delta \leqslant m \leqslant n$, $m \geqslant n^{\varepsilon}$ and $\varepsilon > 0$), and constants $\beta > 2$ and $\theta > 0$ thus having a performance which is better than the performance of all previously proposed synchronizers. Moreover, by chosing $m = \Delta$, for networks with $e = O(\Delta^{\theta})$, we obtain a synchronizer with constant overhead in communication and time. Also by choosing $m = n$, we obtain a synchronizer with constant message complexity.

We assume (as in [6]) that our networks have individual site clocks which run *at the same rate* but are *not* necessarily synchronized. Our networks have transmission delays that are random variables "with memory" (less variable than the exponential). Note that the assumptions about the distribution of the transmission delays affect *only the performance* of our synchronizer. Its correctness is guaranteed independently of those assumptions.

2. The optimistic synchronizer

We assume that a processor has been elected leader, and that T is a spanning tree of the network. Each processor knows which of its edges belong to T.

In the sequel, m is the protocol's parameter, and α, β, θ are constants that will be determined later. When we say x time units, we assume that they are measured according to the processors' clocks (anyone of them, since they all run at the same rate). The synchronizer is similar to the synchronizer of [5, 6]; it has two operating modes. In the *steady-state mode*, a processor executes a step of the synchronous algorithm by performing the following *phase*.

```
begin
  – wait $W=(\alpha+\beta)\log m$ time units;
  – read messages that arrived and originated from neighbors within the previous
    phase;
  – compute, as the simulated synchronous algorithm requires;
  – send messages to neighbors for the next phase, as the simulated algorithm requires;
end
```

The phase is a *tentative* synchronization method. Note that in each phase, each node just waits W time units and does not use any other messages than messages of the algorithm to be simulated received so far. The messages have to be *tagged* by *phase number*. If a message of an older phase arrives (late) at a node, then the node just sets an ERROR flag locally.

While a processor is in steady-state mode, it repeats the phase. After repeating the phase $k=m^{\theta}$ times, the processor enters the *audit mode*, which has the goal of backtracking the computation in case an error has occurred in any of the phases since the last audit mode, namely, during the current *epoch*. The audit mode consists of three stages: *local test*, *global test*, and *restarting*. In the local test a processor finds out if a message which has not yet arrived caused it to simulate a phase of the synchronous algorithm incorrectly. In the global test the leader determines if any processor detected an error during the previous stage, and broadcasts this information. In the restarting, the processors either proceed to the next epoch or backtrack and repeat the last one, as instructed by the leader. A more detailed description of the stages follows.

2.1. *Local audit*

When a processor enters the local audit mode, it finds out if it has executed any phase incorrectly during the current epoch. This test can be performed as follows. If

ERROR is set, then some step has been performed incorrectly. If ERROR is not set, a mistake could have been performed only in case a late message has not yet been received. To check this, each processor sends an END-EPOCH message to every neighbor, and waits for an END-EPOCH message, from every neighbor. The message sent from u to v includes the number of times u sent a message to v during the current epoch (messages may arrive out of order due to our assumption of independent stochastic delays, also the last phase has to be checked). When v has received an END-EPOCH message from every neighbor, it knows whether it has received all the messages of the current epoch. If any of the messages did not arrive on time, v sets its ERROR flag, and waits for any messages that have not arrived yet. This concludes the local audit. Thus this stage serves also the purpose of cleaning the links from any messages.

2.2. *Global audit*

At this point some processors could have the ERROR set, while others did not detect any error locally. After the global audit, if at least one processor had the ERROR set after the local audit, then all the processors will have the ERROR set. The leader starts the global audit broadcasting along the tree an AUDIT message. After relaying the AUDIT message, a processor waits for an answer message from each son in the tree. The answer will be either a YES (in the case of the ERROR flag set) or a NO otherwise message. If the processor receives at least one YES message or if it has its ERROR set, it answers a YES to its parent and sets its ERROR flag. When the leader receives an answer from each son, it sets its ERROR if it received at least one YES message. At this point no messages are in transit in the network. Then the leader starts a second wave of messages: it broadcasts a BACKTRACK message if its ERROR flag is set or else it broadcasts a RESTART message. If a processor receives a BACKTRACK message it sets its ERROR flag. This completes the global audit stage with no messages in transit in the network and either every processor has its ERROR flag set or no processor has its ERROR flag set.

Alternatively a node, if it had its error flag set, it may send a YES to its parent without propagating the AUDIT message below it. This will effectively allow the root to backtrack or proceed in to the next phase sooner. We, however, prefer the first approach for clarity reasons.

2.3. *Restarting*

Once the auditing test is completed, the leader invokes the following distributed restart algorithm (similar to the initialization mechanism of [5, 6]). The leader sends to each of its neighbors a START-EPOCH message. When a processor receives a START-EPOCH message for the first time it sends it out on every link. If ERROR is set it unsets it and rolls back the computation to the previous epoch. Else, if ERROR is not set it commits the epoch. Then the processor enters steady-state mode. Observe that the START-EPOCH messages propagate in the whole network (not only the tree) via flood. The aim is to guarantee that neighboring processors enter steady state more or less at the same time.

2.4. Improvements on the algorithm

Several improvements on the algorithm are possible. Note that if at least one processor sets its flag to ERROR, then every processor in the network rolls back the computation to the beginning of the epoch. This can be improved by appending to the START-EPOCH messages, information including the number of the latest correctly executed phase (i.e. this will allow roll back to the latest correctly executed phase). This information can be gathered by the leader.

It is possible also to modify the algorithm so that no leader is needed. The global audit needs to be modified. Instead, each processor propagates a wave with an AUDIT message including its own i.d. When it gets the answers from each neighbor it knows if at least one processor has its ERROR set. It then enters the restarting stage and broadcasts the START-EPOCH messages. A processor receiving such a message for the first time relays it immediately. Hence a single flood of START-EPOCH messages is propagated, perhaps started by different processors. Yet neighboring processors restart steady state approximately at the same time. Observe that the difference between the times on which two neighbors re-enter steady-state mode is bounded by the message delay of a START-EPOCH message.

3. Correctness and complexity of the optimistic synchronizer

3.1. Outline

From the synchronizer's protocol, it is clear that an epoch which has been incorrectly performed (because some message arrived too late) will be rolled back by every processor. Also, no message of previous epochs remain in the network. If the window W is big enough, then too many errors will not occur, and an epoch will be eventually *committed* when the computations of all nodes during the epoch were correctly done. Thus our synchronizer is correct (it never commits erroneous computations).

Intuitively, if at the beginning of each phase of a node the node and its neighbors are "approximately synchronized" then they will remain so, at the end of the phase (with high probability, depending on a successful selection of W). The audit mode (and the related initialization phase) serve two purposes: To make all nodes "approximately synchronized" and to preserve the correctness of the computation (by a *definite* protocol).

3.2. Complexity analysis

We analyze here the protocol as presented with the leader version (the no leader version has a comparable performance, but the epochs are implicitly committed). For the performance analysis we use the fact that the delays are random variables with memory, of mean at most $1/\lambda$ (λ a parameter). Intuitively, a random variable is with memory if it is "new better than used" in expectation. Many natural distributions belong to this class, such as normal, uniform, and exponential. As we shall now see, the exponential is the one that produces the worst performance of the synchronizer. Therefore, the complexity analysis will be done assuming that the delays are exponentially distributed with mean $1/\lambda$.

Definition (*see also Ross* [10]). A random variable x is called a random variable *with memory* if

$$\forall a \geqslant 0 \quad E(x-a/x>a) \leqslant E(x),$$

where $E(x)$ is the expected value of x. As the reader may notice, we include the exponential random variable in this definition. This is done for reason of uniformity of presentation of Theorem 3.1 and Facts 1 and 2. We adopt this extension of our terminology to the exponential despite the fact that the exponential is called a "memoryless" distribution in probability theory, since it is a limit distribution in our case.

Definition. A random variable x is called *less variable* than a random variable y if, for all increasing convex functions h,

$$E(h(x)) \leqslant E(h(y)).$$

We denote this by $x \leqslant_v y$.

Fact 1 [10]: Let x be a random variable with memory and y be an exponential random variable of same mean. Then $x \leqslant_v y$.

Fact 2 [10]: If $x_1, x_2, \ldots, x_n$ are independent random variables and $y_1, y_2, \ldots, y_n$ are independent random variables and $x_i \leqslant_v y_i$ then $g(x_1, x_2, \ldots, x_n) \leqslant_v g(y_1, y_2, \ldots, y_n)$ for any increasing convex function g which is convex in each of its arguments.

Theorem 3.1. *The probability of an error during a phase of a node is maximized when the message delays are exponential random variables, among all possible delay distributions with memory.*

Proof. An error occurs when the maximum of the message delays from the neighbors (*plus* the max "slack" in initial neighbor synchronization) exceeds W. Since the max function (and the *plus*) are convex, by Facts 1 and 2, all moments are maximized when the delays are exponential random variables. □

Corollary 3.2. *The exponential delays of messages provide the worst-case performance of the optimistic synchronizer, among all possible distributions of delays (of the same mean) which are random variables with memory.*

Theorem 3.3. *For each $\beta > 2$ and $\alpha > 0$ there is a $\gamma > 0$ such that the following holds: Assume that all the neighbors of a node v, and v itself, finish phase i within a time interval of $\alpha \log m$ (where α is a positive constant). Then, all the messages that are sent to v at the end of phase i will be received within the window W of phase $i+1$ of v, with probability $\geqslant 1 - m^{-\beta}$, provided that W is at least $\gamma \log m$, where $\gamma = \alpha + \beta/\lambda$.*

Proof. Let t_0 be the instant at which the last phase i of v's neighbors finishes. In the worst case, all neighbors finish their phase i at t_0 (else they finish earlier).

Let $D_1, D_2, \ldots, D_{m'}$ be the message delays of the messages that were sent to v at the end of phase i.

Let $D = \max\{D_1, D_2, \ldots, D_{m'}\}$.

Clearly m' is at most equal to the number of v's neighbors and hence $m' \leqslant \Delta$.

$$\text{Prob}\{D \leqslant x\} = \text{Prob}\{\forall D_i,\, D_i \leqslant x\} = \prod_{i=1}^{m'} \text{Prob}\{D_i \leqslant x\}$$

because of independence. Since in the worst case all D_i's are exponential, the above probability (of small delay) is minimized when all D_i's are exponential. Thus,

$$\text{Prob}\{D \leqslant x\} \geqslant (1 - e^{-\lambda x})^{m'},$$

where λ is the rate of the exponential.

If we want $\text{Prob}\{D \leqslant x\}$ to be $\geqslant 1 - m^{-\beta}$ then it is enough to have

$$(1 - e^{-\lambda x})^{m'} \geqslant 1 - m^{-\beta} \Rightarrow e^{-\lambda x} \leqslant 1 - (1 - m^{-\beta})^{1/m'}$$

$$\Rightarrow x \geqslant \frac{1}{\lambda} \ln\left(\frac{1}{1 - (1 - m^{-\beta})^{1/m'}}\right).$$

But

$$\left(1 - \frac{1}{m^\beta}\right)^{1/m'} \geqslant 1 - \frac{1}{m^\beta m'} \geqslant 1 - \frac{1}{m^\beta},$$

since $m' \geqslant 1$. Thus,

$$1 - \left(1 - \frac{1}{m^\beta}\right)^{1/m'} \leqslant \frac{1}{m^\beta} \Rightarrow x \geqslant \frac{\beta}{\lambda} \log m.$$

Thus, if we pick a $\gamma \geqslant \alpha + \beta/\lambda$, then all the messages from the various phases i of v's neighbors will indeed arrive at v during the window $W = (\alpha + \beta/\lambda) \log m$ of v's phase $i+1$, with probability $\geqslant 1 - m^{-\beta}$. □

Corollary 3.4. $\exists \beta_0 > 0$: $\forall \beta > \beta_0$, *if all phases i finish correctly within an interval of* $O(\log m)$ *time then all phases $i+1$ will finish correctly within an interval of* $O(\log m)$ *time with probability at least $1 - m^{-\beta}$.*

Proof. Since all clocks rates are the same and there are no drifts, the finishing time of phase $i+1$ of any node happens after $W+x$ steps from the finishing time of phase i of the same node. Here x is the time interval (same for all nodes) to: (1) read messages (pad if necessary), (2) do one step of synchronous algorithm and (3) send messages (pad if necessary) (x is the same for all nodes due to the synchronous algorithm). Thus the finishing times of phase $i+1$ of all nodes are just translated by the same amount from the finishing times of phase i of each corresponding node. Hence the assumption of Theorem 3.3 holds for all phases. Therefore, it is clear that

$$\text{Prob}\{\exists \text{ phase } i+1 \text{ finishing incorrectly}\}$$
$$\leqslant \sum \text{Prob}\{\text{incorrect finishing of a phase } i+1\} \leqslant \sum m^{-\beta_0}$$

and choose β_0 high enough: $nm^{-\beta_0} \leqslant m^{-\beta}$. □

In the sequel, let $k=m^{\theta}$, $\theta>0$.

Theorem 3.5. *For each $\beta_1>3$ and $\alpha_1>0$, $\exists\gamma>0$: If an epoch of the network starts in such a way that all starting moments of neighbor nodes are in an interval of size $\alpha_1 \log m$ ($\alpha_1>0$ a constant) then all nodes will finish the epoch correctly with probability at least $1-m^{-\beta_1}$, provided $W=\gamma \log m$.*

Proof. Let $\beta=\beta_1+\theta$. Choose $\gamma \geqslant \alpha_1+\beta/\lambda$. Let E_j be the event "phase j of the epoch finishes correctly provided that all nodes start it within an interval of $\alpha_1 \log m$".

Then, if $E=\bigcap_{j=1}^{k} E_j$, we wish to find the Prob$\{E\}$. But

$$\text{Prob}\{\bar{E}\}=\text{Prob}\left\{\bigvee_{j=1}^{k} \bar{E}_j\right\} \leqslant \sum_{j=1}^{k} \text{Prob}\{\bar{E}_j\} \leqslant km^{-\beta}=m^{-(\beta-\theta)}=m^{-\beta_1}.$$

Thus, Prob$\{E\} \geqslant 1-m^{-\beta_1}$. □

Theorem 3.6. *For each $\alpha>0$ $\exists\beta>0$: Each epoch of our synchronizer starts in such a way that all starting moments of neighbor nodes are within an interval of size $\alpha \log m$ ($\alpha>0$ is an appropriate constant) with probability at least $1-m^{-\beta}$.*

Proof. The starting time of the last of any set of neighbors to start is at most the maximum of m exponential independent random variables of mean $d=1/\lambda$ (because of the flood protocol by which the leader starts or restarts an epoch). The probability that this maximum can exceed $\alpha \log m$ is at most $m^{-\beta}$ for some β depending on α (proof as in Theorem 3.3). □

Theorems 3.5 and 3.6 show the following corollary.

Corollary 3.7. *Each epoch terminates correctly with probability at least $1-m^{-\beta}$ and β can be controlled by adjusting the window size W.*

From Corollary 3.7 and from the fact that the mean value of a geometric random variable Y of density $\mathrm{Prob}\{Y=i\}=(1-p)^{i-1}p$ is bounded above by $1/p$, we get the following corollary.

Corollary 3.8. *The mean number of unsuccessful repetitions of an epoch before commit is bounded above by* 2.

Proof. Since an epoch fails with probability $\leqslant n^{-\beta}$ just put $p=1-m^{-\beta}$. □

Thus we get our main result.

Theorem 3.9. *The amortized (over an epoch) expected number of synchronizer messages (per synchronous step of the simulated algorithm) is* $\mathrm{O}(e/m^{\theta})$. *The amortized (over an epoch) expected delay of the synchronizer (per "synchronous" step of the simulated algorithm) is* $\mathrm{O}(\beta \log m)$, *for a constant* β *and* $\Delta \leqslant m \leqslant n$.

Proof. Select a $\theta>0$ $(k=m^{\theta})$, for $\theta=\beta-\beta_1$ and $\beta_1>3$. The number (expected) of messages of the synchronizer per epoch is $\mathrm{O}(e)$ ($e=$ number of network edges), $2e$ for END-EPOCH messages, n for AUDIT, e for YES or NO, and e for BACKTRACK or RESTART messages by protocol description, and by the fact that no messages of the synchronizer are used in phases. Thus the amortized number is $\mathrm{O}(e)/k=\mathrm{O}(e/m^{\theta})$, for the constant θ.

The total delay per epoch per node is at most $\gamma k \log m$ (for the phases of the epoch, where γ is as in Theorem 3.3) with probability at least $1-m^{-\beta}$, plus the delay of the commit protocol. The mean delay of the commit protocol is $\mathrm{O}(1/\lambda)$. In this delay we must add the contribution of the global restart delay which is $\mathrm{O}(1/\lambda)$ on the average. If we examine a large but not infinite sequence of epochs of our protocol then for most of the epochs the neighboring nodes remain approximately synchronized (in the sence that corresponding starting times are within $\mathrm{O}(\log m)$). However, this will eventually be violated (after m^{β} epochs on the average). In that case the next epoch will be erroneous due to the delays. However, the subsequent restart by the leader (by using the spanning tree and the flood) will again approximately synchronize the neighboring nodes with probability at least $1-m^{-\beta}$. In this stochastic process, we take amortized values over this large number of epochs. Then we take expected values. Due to ergodicity then the total delay is (expected) $\mathrm{O}(k \log m)$ and the amortized value is $\mathrm{O}(\log m)$. □

4. Future work and extentions

We are currently extending our optimistic synchronizer to work for networks whose node clocks are not accurate but can be synchronized by another protocol.

Our optimistic synchronizer presented so far has to know an appropriate window size W in order to work efficiently, since the multiplication constant of W depends on the mean message delay. The mean message delay can be estimated (and adjusted when the protocol starts doing a lot of restarted epochs) by the following simple protocol.

4.1. Mean delay estimation protocol for node v

(1) For each neighbor w of v, v sends a "count delay" message, and stores the sending time $t(v, w)$. If v receives such a message, it sends it back to its originator as soon as it receives it.

(2) When v receives the count delay message back, it notes the receipt time (according to v's clock) $t'(v, w)$. Let $d(v, w) = t'(v, w) - t(v, w)$.

(3) The above is repeated g times. Let $d_i(v, w)$ the estimate of each time.

Then

$$\tilde{d} = \frac{d_1(v, w) + d_2(v, w) + \cdots + d_g(v, w)}{2g}.$$

With the delay estimation protocol our synchronizer can be applied to networks where mean message delays vary with time, and where message delay distributions are not the same in each neighbor (then one has to use the *largest* estimated mean delay in the formula for W).

Acknowledgment

P. Spirakis wishes to thank Z. Kedem and K. Palem for insightful comments on rollback techniques, and C. Bouras for many discussions on the general topic of average case analysis of distributed protocols.

References

[1] B. Awerbuch, Complexity of network synchronization, *J. Assoc. Comput. Mach.* **32** (1985).

[2] B. Awerbuch and D. Peleg, Network synchronization with polylogarithmic overhead, in: *Proc. IEEE FOCS* (1990).

[3] F.C. Commoner, W. Holt, S. Even and A. Pnueli, Marked directed graphs, *J. Comput. System Sci.* **5** (1971).

[4] K.M. Chandy and L. Lamport, Distributed snapshots: determining global states of distributed systems, *ACM Trans. Comput. Systems* **3** (1985).

[5] E. Even and S. Rajsbaum, Lack of global clock does not slow down the computation in distributed networks, TR No. 522, Department of Computer Science, Haifa, Israel, 1988. The first part of this paper appeared also with the title "Unison in Distributed Networks" in: R.M. Capocelli, ed., *Sequences, Combinatorica, Compression, Security and Transmission* (Springer, Berlin, 1990).

[6] E. Even and S. Rajsbaum, The use of a synchronizer yields maximum rate in distributed networks, in: *Proc. 22nd ACM STOC* (1990).

[7] Z. Kedem, K. Palem, A. Raghunathan and P. Spirakis, Combining tentative and definite executions for very fast dependable parallel computing, in: *Proc. ACM STOC* (1991).
[8] D. Peleg and J. Ullman, An optimal synchronizer for the hypercube, *SIAM J. Comput.* **18** (1989) 740-747.
[9] S. Rajsbaum and M. Sidi, On the average performance of synchronized programs in distributed networks, in: *Proc. WDAG* (1990).
[10] S.M. Ross, *Stochastic Processes* (Wiley, New York, 1983).
[11] J.C. Wyllie, The complexity of parallel computations, Ph.D. dissertation, Department of Computer Science, Cornell University, Ithaca, New York, 1981.