A Synchronizer with Low Memory Overhead, Extended Abstract

Lior Shabtay
Computer Science Department
Technion IIT
Haifa, Israel 32000

Adrian Segall
Computer Science Department
Technion IIT
Haifa, Israel 32000

Abstract

A new message-delaying version of synchronizer γ, named ζ, is presented. Synchronizer ζ ensures that original-protocol messages received by a node from nodes in the same cluster are never early, *and thus, no buffers for their temporary storage are necessary. Only original-protocol messages on edges leading to nodes of other clusters (external edges) may be* early. *The z-partition algorithm is introduced to reduce the number of external edges connected to each node, thus reducing the memory overhead of ζ. For an arbitrary z, this algorithm ensures that the external degree of each node is no more than $\lfloor \frac{|V|}{z} \rfloor - 1$. The z-partition algorithm increases the time complexity of ζ to $O(z + \log_k |V|)$ per pulse. The tradeoff between memory overhead and time complexity achieved by the z-partition algorithm is optimal.*

1 Introduction

This paper deals with distributed protocols in two network models: the *synchronous* model and the *asynchronous* model. In the *asynchronous model*, nodes perform operations only upon receiving a message from some neighbor or from the outside world. At that time, the node processes the message, performs local computations, and may send messages to some or all of its neighbors. All local actions are performed atomically. Messages sent by a node to any of its neighbors are received in a FIFO order within a finite undetermined time. Messages are processed at each node in the received order, even if they were received from different links. The contents of any received message is not available to the code activated by later received messages, unless it was saved when received in the memory allocated to the distributed protocol.

The *communication complexity* of an asynchronous protocol for a network N is defined as the maximum number of messages sent during an execution of the protocol. The *time complexity* of an asynchronous protocol is defined as the largest time needed to complete an execution of the protocol, assuming the message delay over all links in the network is bounded by one time unit.

The *synchronous model* assumes that all link delays are bounded by some quantity referred to as a time unit. Pulses are generated synchronously at all nodes in the network at time unit intervals. Messages are sent only at pulse ticks, and thus arrive at the destination node before the next pulse. Operations are performed by a node only at the time of a pulse or when receiving a message. When a node receives a message, it processes the message and performs local computations. At the time of a pulse, the node may perform local computations and in addition, it may send messages to some or all of its neighbors. All local actions are performed atomically.

This synchronous model is *message-and-pulse driven*, meaning that the messages are processed only upon arrival, and are not available to the code at any later time. In particular, the contents of any received message is not available to the code at the time of a pulse, unless saved in the local variables when received. The *message-and-pulse driven* model is used in previous works concerning synchronizers, like [1], [2], [4], [5], [13], [14], [16], [19], [20], [21].

We also assume that in a synchronous network, at the time of pulse(n), each node sends at most one message to each neighbor. This assumption has been used in previous works on synchronizers ([1], [2], [5], [6], [7], [8], [9], [10], [13], [16], [19], [20], [21]), and is reasonable because multiple sent messages can be simulated by packing them into one message.

Synchronizers are tools for transforming protocols written for the synchronous model into protocols that run on an asynchronous network. The synchronous protocol will be referred to as the *original protocol*. The asynchronous protocol created by the synchronizer generates a sequence of pulses at each node. The pulses occur asynchronously at different nodes. At each pulse, the nodes perform the original-protocol pulse code and send messages which are identical to the original-protocol messages. In certain circumstances, slight changes in the pulse code and/or in messages are allowed (see [19]).

The methodology of synchronizers was introduced in [1], where three synchronizers were presented: the α synchronizer, with an overhead of $O(|E|)$ in communication complexity and $O(1)$ in time complexity per pulse, the β synchronizer with an overhead of $O(|V|)$ in communication and $O(D)$ in time complexity per pulse (when D is the diameter of the network), and the γ synchronizer, which enables trade-off between the above complexities. Other types of synchronizers can be found in [6], [7], [8], [9], [12], [13], [16], [17], [19] and [20]. Applications and other aspects of syn-

1063-6927/94 $03.00 © 1994 IEEE

chronizers can be found in [2], [4], [5], [14], [15] and [21].

All synchronizers ensure that a node may perform a new pulse when it knows that it has received all original-protocol messages sent to it by its neighbors at the former pulse. However, most synchronizers allow original-protocol messages sent by a node at a given pulse to arrive at a neighbor node *before* the time when the latter has performed that same pulse. We refer to such messages as *early messages*. On the other hand, in a synchronous model, the nodes perform the pulse simultaneously, and every message sent at a certain pulse arrives at the neighbor node after the pulse has been performed. Thus, if no special care is taken, the simulation of the synchronous algorithm, created by using a synchronizer, may allow erroneous executions. We refer to original-protocol messages which are not *early* as *timely* messages.

One way to solve this problem is suggested in [1], [7], [8], [10], [13], [19], [20] and is referred to as *message delaying*. The idea is that when a message is *early*, it is not processed immediately. Instead, such messages are stored in memory and processed only after the pulse is performed at the node. This method is intuitive and simple to implement. It requires buffering at most one message per link at a time, and thus the required number of buffers is equal to the degree of the node. The size of each buffer must correspond to the longest possible original-protocol message. The purpose of this paper is to investigate ways to reduce this amount of memory.

In Sec. 3, we present synchronizer ζ, which is a new version of synchronizer γ. When using this synchronizer, the edges connected to each node are divided into two groups, one on which *early* messages may be received and one on which *early* messages are never received.

In synchronizers γ and ζ, the network is partitioned into clusters. Edges that connect two nodes in the same cluster are referred to as *'internal edges'*. Edges that connect nodes belonging to different clusters are called *'external edges'*. Synchronizer ζ ensures that original-protocol messages received from internal edges are *timely*, so that *early* original-protocol messages may be received only from external edges. Thus, the number of buffers that need to be reserved at each node for *early* messages is the *external degree* of the node, namely the number of external edges connected to the node. If the partition of the network is such that the external degree of the nodes is minimized, the required number of buffers is minimized too.

In [21], we introduce the notion of *memory overhead* of synchronizers. The memory overhead of a synchronizer for a network N and a synchronous (original) protocol P is defined as the maximum over the nodes in N of the memory needed for the synchronizer variables and buffers when combined with P. The memory overhead of a synchronizer is expressed by using the following parameters:

1. S: the maximum over the nodes in the network, of the amount of memory used for storing the synchronous-protocol variables.
2. m: the length of the largest original-protocol message.
3. Network topology parameters: $|V|$, $|E|$, the maximum node degree over the network d, etc.

In [21], we discuss the known synchronizers and their memory overhead.

The memory-overhead of message-delaying synchronizers is composed of two parts: the memory required for the synchronizer protocol and the memory needed for saving the delayed messages. The maximum number of messages that may be delayed at the same time by any synchronizer is $(d-1)$. Thus, the amount of memory needed for saving delayed messages is $(d-1)m$. The amount of memory needed for the synchronizer protocol in γ is $O(d)$, thus the memory overhead of the message-delaying version of γ is $(d-1)m+O(d)$.

In this paper we assume that the maximum degree of a node in a network of $|V|$ nodes is bounded only by $|V|-1$. This assumption is not always true, since in many cases each node has a bounded number of links that can be connected to it. However, in such cases, $|E| = O(|V|)$ and thus synchronizer α is a better choice than γ or ζ since it allows $O(|V|)$ communication and $O(1)$ time complexity for such cases. When assuming that d is bounded only by $|V|$, the memory overhead of synchronizer γ is bounded by $(|V|-1)m+O(|V|)$. Observe that for some original-protocols (e.g. the synchronous protocol presented in [21]), m can be as large as S. For such cases, the memory overhead of γ is $(|V|-1)S+O(|V|)$.

As shown in Sec. 4, if ζ uses the original partition algorithm of γ [1], the memory overhead is the same as in γ, namely $(|V|-1)m+O(|V|)$. The reason is that the external degree of the nodes can still be as large as $|V|$. In order to reduce the external degree, we present in Sec. 5 a new partition algorithm, referred to as the *z-partition algorithm*. This algorithm is given two parameters, k and z, and ensures that the external degree of the nodes does not exceed $\lfloor \frac{|V|}{z} \rfloor - 1$. Therefore, the memory overhead of ζ when using the z-partition algorithm is $(\lfloor \frac{|V|}{z} \rfloor - 1)m + O(|V|)$. When using this partition algorithm, the time complexity of ζ is $O(z + \log_k |V|)$ per pulse, and the communication complexity is $O(k|V|)$ per pulse. This gives a tradeoff between the time complexity and the memory overhead of ζ. However, selecting $z \leq \log_k |V|$ we obtain a reduction of the memory overhead by z, while the order of time complexity is not increased. We prove in [22] that the tradeoff between the time complexity and the memory overhead achieved by the z-partition algorithm is optimal.

2 Preliminaries

2.1 The memory model

The model definition (Sec. 1, first paragraph) says that a node can process an earlier received message after processing later received messages, only if the earlier message was saved in the memory allocated for the protocol algorithm. In other words, we *exclude* the possibility that the processor logically delays message processing without providing additional storage.

To see that this stipulation of the model is in accordance to reality, consider two node configurations, one with a common queue for messages received from all links, and one with a queue designated for each link. In the first configuration, the interface between the algorithm and the lower layer (DLC) uses one fifo queue of messages for all links. Received messages are added at the tail of the queue when received, and the algorithm may process only the message at the head of the queue. In this configuration, the only way to process an earlier received message after processing later received messages is if the algorithm copies the earlier message into an additional buffer that belongs to the algorithm.

A different version of this configuration is when the algorithm is allowed to process messages beyond the queue head. Apparently, this version allows the algorithm to delay received messages without providing additional storage. The algorithm may simply process later received messages, leaving the earlier received messages in the buffers at the head of the queue. However, this requires enlarging the queue with amount of buffers that is equal to the number of simultaneously delayed messages. This amount of memory should be taken into account when calculating the memory requirements of the distributed algorithm.

In the second configuration, the interface between the algorithm and the lower layer uses a number of fifo queues, one for each link. In this case, an earlier received message can be processed after later received messages, if these messages were received from different links. However, if the interface to the lower layer is shared by several algorithms, each algorithm must process each received message immediately and remove it from the queue head so that messages aimed to other algorithms are not delayed. Therefore, in this configuration also, processing postponement requires additional storage.

In the rest of this paper, when calculating the memory overhead of synchronizers, we take into account the buffers used for delaying messages and the memory needed for the synchronizer protocol variables.

2.2 Safety

In a combined protocol, a node i is said to be *safe* with respect to pulse(n), if all original-protocol messages sent by node i at time $t_i(n)$ have already been received by the respective neighbors ($t_i(n)$ is defined as the time at which node i performs pulse(n)). In order to allow the nodes to know when they are safe, each node that receives an original-protocol message is required to send back an explicit acknowledgment message.

Most synchronizers are based on the observation that a node i can perform pulse(n) when all its neighbors are safe with respect to pulse($n-1$), since this means that node i has received all the messages sent to it at pulse($n-1$).

2.3 Synchronizer ϵ

Since synchronizer ϵ [19] is used as a building block in the construction of synchronizer ζ, we give here a brief description of ϵ. In synchronizer ϵ, an initialization phase creates a directed spanning tree for the network graph. After performing pulse($n-1$), SAFE messages propagate along the spanning tree from the leaves to the root. Each node sends a SAFE message to its parent as soon as it is *safe* and has received SAFE messages from all its children. By the end of this process, the root node knows that all the nodes in the network are safe with respect to pulse($n-1$) and ready to perform pulse(n).

At this point, the root node sends AWAKE messages to all its children in the spanning tree and performs pulse(n). Other nodes send AWAKE messages to all their children in the spanning tree and perform pulse(n) upon receiving the first of the two following messages: an AWAKE message or an original protocol message which was sent at pulse(n). In case the pulse was activated by the receipt of an original-protocol message, this message is processed immediately after the pulse is performed. This synchronizer differs from synchronizer β [1] in which a node performs pulses only upon receiving AWAKE messages.

Observe that no *early* messages need to be saved since all messages are processed by the code activated by their arrival and before any other messages are processed (See Sec. 2.1). However, the code is performed in the right order — when an *early* message activates the pulse, the pulse is performed before the message is processed. The memory overhead of ϵ is $O(d)$, its time and communication complexities are $O(D)$ and $O(|V|)$ per pulse respectively.

2.4 Synchronizer γ

In synchronizer γ [1], an initialization phase creates a partition of the network into clusters. The partition is defined by a spanning forest of the network. Each tree in the forest defines a cluster of nodes. Between every two neighboring clusters, one preferred link is selected. Inside each cluster there is a *leader node*.

After each pulse, SAFE messages are converged along each cluster tree from the leaves to the leader node. At this point, CLUSTER_SAFE messages are broadcast on the edges and preferred links of each cluster spanning tree, telling all neighboring clusters that this cluster is safe. The information that neighboring clusters are safe is brought to the cluster leader by means of CLUSTER_READY messages which converge along each tree, and then the leader node initiates a broadcast of AWAKE messages along the cluster tree, which cause the nodes in the cluster to perform the next pulse.

The communication and time complexities of γ are $O(k|V|)$ and $O(\log_k(|V|))$ per pulse respectively (k is a parameter given at initialization time, see Sec. 2.5).

2.5 The partition algorithm of γ

The initialization phase of γ is called a *partition algorithm*. This algorithm creates a partition of the network into clusters, builds a spanning tree for each cluster, and selects a preferred link between each two neighboring clusters.

In this partition algorithm, clusters are built one by one. Each time, a new cluster is built from nodes in the remaining graph (nodes which are not in one of the previously built clusters). This is done by selecting a leader node and then creating the cluster from nodes

in the remaining graph that are around this leader node.

The job of creating the cluster around a given leader node is performed by a procedure named Cluster_Creation. The operation of this procedure is controlled by a constant k ($2 \leq k \leq |V|$) given to the partition algorithm when initiated. Cluster_Creation operates in the following way: the selected leader node triggers an execution of a BFS protocol in the remaining graph. The nodes at each new BFS layer join the cluster as long as the number of nodes in the new layer is at least $(k-1)$ times the total number of nodes in all previous layers.

Definition 1: *(E_p, H_p)*
Given a partition P of a network into clusters with spanning trees and with a preferred link between each two neighboring clusters, E_p is defined as the number of preferred links plus the total number of edges in all cluster spanning trees. H_p is defined as the maximum height of a cluster spanning tree.

The time complexity of γ when using a partition P is $4H_p$. The Cluster_Creation procedure ensures that $H_p \leq \log_k |V|$, therefore, the time complexity of γ is $4 \log_k |V|$. The communication complexity of γ when using a partition P is $4E_p$. The Cluster_Creation procedure ensures that $E_p \leq k|V|$, therefore, the communication complexity of γ is $4k|V|$.

The partition algorithm is initiated by *one node*. If a leader is not present in the network a-priory, a leader election procedure [11], [3] must be used. This node starts the algorithm by calling the Cluster_Creation procedure, that ends at the same node after creating a cluster led by it. At this point the node calls the Search_For_Leader procedure, which searches for a leader of a new cluster. The procedure ends at a free node, if any. This free node calls Cluster_Creation which creates a cluster around it, then it calls Search_For_Leader, and so on. The Search_For_Leader procedure creates a DFS tree of clusters, where the parent of a cluster C is defined as the cluster from which the leader of C was discovered. One more procedure, named Preferred_Link_Election, is used by the algorithm. This procedure selects the preferred links between every two neighboring clusters.

3 Synchronizer ζ

In this section we discuss a new synchronizer, named ζ, which is a message-delaying version of synchronizer γ that provides a tradeoff between the time and communication complexities and the memory overhead. Synchronizer ζ provides a significant reduction in the memory overhead for a small penalty in time and communication complexity.

Definition 2: *(internal and external edges)*
We say that an edge is internal *or* external *depending on whether it connects nodes in the same or in different clusters.*

In synchronizer ζ, we suggest altering the mechanism by which the leader node causes the nodes in the cluster to perform a new pulse. The mechanism is altered such that original-protocol messages sent over internal edges are always *timely*. Thus, nodes do not ever need to save messages arriving from internal edges, but still need to save *early* messages arriving from external edges.

In synchronizer γ, after each pulse, the nodes perform a protocol to inform each leader node when all the nodes in its cluster are ready to perform the next pulse. The leader node then causes the nodes in the cluster to perform a new pulse by broadcasting an AWAKE message over the cluster tree. In synchronizer ζ we suggest to replace this broadcast with the mechanism of synchronizer ϵ [19]. Since ϵ ensures that no original-protocol messages arrive *early*, we ensure that original-protocol messages sent by nodes to other nodes in the same cluster are always *timely*. This mechanism is described here briefly and a description of synchronizer ϵ is given in Sec. 2.3.

When the leader node knows that all the nodes in its cluster are ready to perform the new pulse, it sends AWAKE messages to its children in the cluster spanning tree and then performs the pulse. Nodes in the cluster perform the new pulse when the first of the following happens: (1) an AWAKE message is received from the parent of the node in the tree, or (2) an original-protocol message sent at the new pulse is received from an internal edge. *Early* original-protocol messages which are received from external edges do not cause the node to perform the new pulse; instead such messages are saved and processed only after the pulse. A node which performs the new pulse sends an AWAKE message to each of its children in the cluster tree before performing the new pulse. If the new pulse was triggered by the receipt of an original-protocol message, the message is processed immediately after performing the pulse.

The protocol requires nodes to be able to distinguish between *timely* and *early* original-protocol messages. When receiving an original-protocol message from an internal edge, the node should perform a new pulse only if the message is *early*. When receiving an original-protocol message from an external edge, the node processes this message immediately if it is *timely*, and saves the message in order to be processed later if it is *early*. One way to distinguish between the two types is by adding the number of the pulse (one bit) as a suffix to each original-protocol message. Two more ways are discussed in [19] and [20].

In order to show that ζ works properly, the following two properties should be proved:

1. nodes do not perform a pulse before they are ready to do so (received all the messages of the former pulse).

2. messages sent at pulse(n) are processed by the receiving node after performing pulse(n) and before performing pulse($n+1$).

In order to show that the first property holds, recall that in ζ, exactly as in γ, the leader node of each cluster sends AWAKE messages and performs pulse($n+1$)

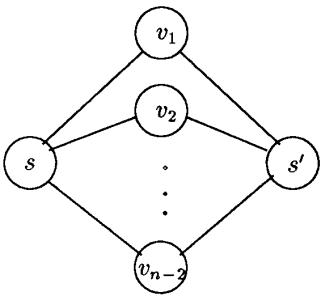

Figure 1: A network with two high degree nodes

only after all the nodes in the cluster and in the neighboring clusters are *safe*. Thus, when the leader performs pulse$(n+1)$, all the nodes in the cluster have already received the messages sent to them at pulse(n). The leader node is the first in the cluster to perform pulse$(n+1)$ and, therefore, the first property holds.

In order to show that the second property holds, recall that original-protocol messages sent to a node i at pulse(n) are received by i between $t_i(n-1)$ and $t_i(n+1)$. Those received on external edges before $t_i(n)$ are delayed and processed only after i performs $t_i(n)$. If the first message received by i on an internal edge is received before $t_i(n)$, it causes i to perform $t_i(n)$ and only after this is done, the message is processed. Thus, all messages sent to node i at pulse(n) are processed after $t_i(n)$ and before $t_i(n+1)$, and the second property holds.

4 The original partition algorithm

Recall that the memory overhead of the ζ synchronizer at a certain node is proportional to the number of external edges connected to it:

Definition 3: *(external degree of a node)*
The external degree *of a node in a partition P of a network N is defined as the number of external edges connected to it, when the partition P is used.*

We seek a partition algorithm that reduces the external degree of the nodes as much possible for each given network.

The partition algorithm [1] used for synchronizer γ does not help to reduce the memory overhead when used to initialize ζ. We show that this claim is true by presenting an example of a network in which a node that has a degree $d = |V| - 2$ also has an external degree of $|V| - 2$.

Consider the network of Fig. 1. Assume that node s initiates the partition algorithm. If $k = |V|$, each node becomes a separate cluster. If $k \leq |V|-1$, the first created cluster contains the nodes s and $v_1, v_2, \ldots v_{n-2}$. In both cases, node s' forms a single-node cluster, and therefore, the external degree of s' is $|V| - 2$. Thus, the memory overhead of ζ is, in this case, identical to the memory overhead of γ and very high $(m \times (|V| - 2) + O(|V|))$.

5 The z-partition algorithm

In this section we present a partition algorithm that, given a parameter z $(2 \leq z \leq |V|)$, reduces the maximum external degree of the nodes to $\lfloor \frac{|V|}{z} \rfloor - 1$. We assume that the nodes know in advance the exact value of $|V|$. If this is not the case, the leader-election procedure [11],[3], which initiates this algorithm, can be used to provide this information without increase of communication or time complexities. For this partition algorithm, we classify the nodes of the network into the following three groups:

1. *red nodes* — nodes that have degree which is larger than $\lfloor \frac{|V|}{z} \rfloor - 1$.
2. *gray nodes* — nodes which are not red, but have a red neighbor.
3. *white nodes* — are not red and do not have a red neighbor.

We also classify the edges in the network into three categories:

1. *red edges* — a red edge is an edge that connects a red node with some other node.
2. *white edges* — connect white nodes.
3. *blue edges* — none of the above.

Our goal in the z-partition algorithm is to ensure that all red nodes have an external degree 0. Thus, no node has an external degree of more than $\lfloor \frac{|V|}{z} \rfloor - 1$ and our goal is fulfilled. This is done by ensuring that all neighbors of a red node are in the same cluster as the red node itself. The z-partition algorithm creates two kind of clusters:

1. clusters consisting of red and gray nodes only. We will call such clusters 'z-clusters'.
2. clusters consisting of white nodes only, created by using the original Cluster_Creation procedure of the original partition algorithm. We will call such clusters 'k-clusters'.

The z-partition algorithm consists of an initialization stage and a partition algorithm, that starts when the initialization ends. The initialization algorithm uses two types of messages: PASS_TYPE and PASS_MEMBERSHIP. A PASS_TYPE($flag$) message contains a flag that tells whether the node that sends the message is red or not. A PASS_MEMBERSHIP ($flag$) message contains a flag that tells whether the node that sends the message should be a member of a z-cluster (red or gray nodes) or of a k-cluster (white nodes).

The initialization stage is initiated by one node s. Node s starts an execution of a PIF (Propagation of Information with Feedback) [18] of PASS_TYPE messages. By the end of this PIF protocol, each node knows which of its neighbors is a red node (if any). Now, each node knows also whether it should be a

member of a z-cluster or of a k-cluster: members of a z-cluster are red nodes and their neighbors (which are the gray nodes).

At this point, node s initiates another PIF, this time of PASS_MEMBERSHIP messages. When this PIF protocol ends, each node knows which of its neighbors belongs to a z-cluster and which belongs to a k-cluster. When the second PIF protocol ends, the initialization stage is over and node s initiates the partition algorithm.

The partition algorithm is a version of the original partition algorithm with an altered Cluster_Creation procedure. The new Cluster_Creation procedure is built of two sub-procedures: z_Cluster_Creation and k_Cluster_Creation. The main procedure calls z_Cluster_Creation if the cluster leader (which is the node that initiates the procedure) is a red or gray node, and calls k_Cluster_Creation if the cluster leader is a white node.

The k_Cluster_Creation procedure operates in a way which is very similar to the operation of the original Cluster_Creation procedure. The only difference is that this procedure builds the cluster from white nodes only. In other words, the procedure uses only white edges and ignores other types of edges.

The z_Cluster_Creation procedure builds the cluster spanning tree by executing a BFS protocol that uses only red edges. The BFS algorithm continues to append new layers to the cluster as long as these layers are not empty.

The Search_For_Leader and Preferred_Link_Election procedures used by the z-partition algorithm are those used by the original partition algorithm presented in [1].

5.1 The time and communication complexities of ζ, using the z-partition algorithm

In order to understand how large the diameter of a z-cluster can be, see Fig. 2. This figure demonstrates the case in which $2z-1$ red nodes form a path in which the first red node shares a neighbor (or more) with the second one, the second red node shares neighbors with the third one, etc. The nodes in this network are joined into one long z-cluster by the z-partition algorithm. This is due to the fact that each red node is in distance of 2 from the preceding red node and the next red node in the path, and all the other nodes are in distance of one from a red node. Since all the red nodes are ordered in a long path, this is the longest z-cluster possible. The diameter of such a cluster is at most $4z-5$.

Recall that the time complexity of ζ per pulse is four times the height of the highest cluster spanning tree (called H_p). The k_Cluster_Creation procedure creates clusters of height $H_p \leq \log_k |V|$ (this is proved in [1]). The highest z-cluster is at most of height $4z-5$. Thus, the z-partition algorithm ensures that $H_p \leq \max\{\log_k |V|, 4z-5\}$. The time complexity of ζ per pulse, when using the z-partition algorithm, is therefore, $4 \times \max\{\log_k |V|, 4z-5\}$.

The communication complexity of synchronizer ζ is four times the number of edges which participate in the synchronizer protocol (called E_p). This contains the edges of the spanning forest, which are at most $|V|-1$, and the chosen preferred links.

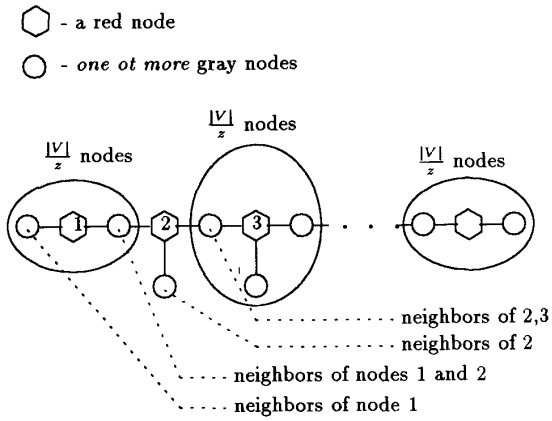

Figure 2: A cluster with diameter $4z-5$

The k_Cluster_Creation procedure ensures that the number of preferred links connecting neighboring k-clusters is at most $(k-1)|V|$. The maximum number of z-clusters is z. There are at most $|V|$ clusters in the whole partition. Thus, the number of preferred links connecting z-clusters to k-clusters or z-clusters, is at most $z|V|$. We conclude that the upper bound on the number of preferred links is $((k-1)+z)|V|$, and therefore, $E_p \leq (k+z)|V|$. The communication complexity of ζ per pulse, when using the z-partition algorithm, is therefore, $4(k+z)|V|$.

These results show that the z-partition algorithm provides a tradeoff between the memory overhead of ζ, and its time and communication complexity: the larger the parameter z given to this partition algorithm, the smaller the memory overhead, and the larger are the communication and the time complexities. For example, selecting $z = \min\{k, \lceil \log_k |V| \rceil\}$, reduces the memory overhead of ζ by factor z (from $(|V|-1)m + O(|V|)$ to $(\lfloor \frac{|V|}{z} \rfloor - 1)m + O(|V|)$), without increasing the time or communication complexities.

5.2 An improved z-partition algorithm

In this section we present an improved z-partition algorithm. The improvement is done by changing the k_Cluster_Creation procedure. in the sequel we show how this improvement reduces E_p to be $O(k|V|+z^2)$, and how this improvement does not change the order of H_p.

In the original k_Cluster_Creation procedure, a new layer of nodes is examined at each pulse. If the new layer contains at least $k-1$ times more nodes than are in all former layers, it joins the cluster; if not, it is rejected.

This k_Cluster_Creation procedure is changed as follows: in the altered procedure, as long as the cluster contains less than z nodes, the new layer is not rejected, no matter how few nodes it contains. In the

case where the new layer contains no nodes at all, while there are less than z nodes in the cluster, this cluster joins the cluster that elected its leader (its parent cluster in the clusters DFS tree). In the sequel, we call the action of joining the parent cluster *'hooking'*.

Observe that in the case where the problematic cluster is the first created by the partition algorithm, it cannot perform hooking. In this case, the cluster remains as it is.

The following discussion refers to a partition created by an execution of the improved z-partition algorithm, on an arbitrary network.

Lemma 1: *The number of preferred links between k-clusters is at most $(k-1)|V|$.*

Proof: The proof of this lemma is similar to the one in [1], since it is not affected by the changes made to k_Cluster_Creation. We count only preferred links from a k-cluster to k-clusters that were formed after it. In this way, each link is counted exactly once.

The k_Cluster_Creation procedure rejects a new layer only if the number of nodes in this layer is less than $k-1$ times the number of nodes already in the cluster. Thus, when a k-cluster which contains c nodes is formed, at most $(k-1)c$ free white nodes are at distance of one hop from it. Therefore, at most $(k-1)c$ of the k-clusters formed afterwards may be its neighbors. The sum of preferred links between k-clusters is thus less than $(k-1)\Sigma c \leq (k-1)|V|$.

Hooking of clusters does not change this outcome, since it only reduces the number of k-clusters and thus the number of preferred links between them. □

Lemma 2: *The number of k-clusters is at most $\lfloor \frac{|V|}{z} \rfloor + 1$.*

Proof: With the altered k_Cluster_Creation procedure, every k-cluster except for the first one *hooks* on to another cluster if it contains less than z nodes. Thus, each of the remaining k-clusters contains at least z nodes and therefore the number of k-clusters cannot exceed $\lfloor \frac{|V|}{z} \rfloor + 1$. Number 1 is added to take into consideration the fact that if the first cluster created is a k-cluster with less than z nodes, it does not hook to any other cluster. □

The number of z-clusters is at most z since each one contains at least $\lfloor \frac{|V|}{z} \rfloor$ nodes. Thus, the number of preferred links between z-clusters is at most z^2, and the number of preferred links connecting z-clusters to k-clusters is at most $z \times (\lfloor \frac{|V|}{z} \rfloor + 1) \leq |V| + z$.

An upper bound for the number of preferred links can be now easily computed: $(k-1)|V|$ links between k-clusters, z^2 links between z-clusters, and $|V|+z$ links connect z-clusters to k-clusters. This sums up to $k|V|+z^2+z$, and when adding at most $|V|-1$ edges of the spanning forest, we gain: $E_p \leq (k+1)|V|+z^2+z-1 < (k+2)|V|+z^2$. The communication complexity of ζ with this partition is $4 \times E_p < 4(k+2)|V|+4z^2$ per pulse, which is obviously better than the complexity achieved by the z-partition algorithm.

In case we select $z \leq \sqrt{|V|}$, the communication complexity becomes $4(k+2)|V|+4|V| = 4(k+3)|V|$, which is of the same order as the communication complexity with the original partition algorithm presented in [1].

Let us first analyze the maximum height of a cluster spanning tree in a partition created by the improved z-partition algorithm, assuming no *hooking* is done.

The z_Cluster_Creation procedure is not changed, thus the maximum height of a z-cluster is $4z-5$. The highest possible k-cluster is created as follows: the first $z-2$ layers contain exactly one node each, creating a cluster of $z-1$ nodes, $z-2$ hops high. The later formed layers must obey the rule that each layer contains $k-1$ times more nodes than all former layers. Thus, no more than $\log_k |V|$ such layers can be added. The height of this cluster is no more than $z + \log_k |V|$.

Observe that a k-cluster C_1 hooks on another cluster C_2 only when there are no free white nodes neighboring C_1. Thus, new k-clusters neighboring C_1 are not created, and no cluster hooks C_1. The height of C_1 is at most z, and no other cluster can hook it, therefore C_1 adds at most z to the height of C_2. In fact, no matter how many k-cluster hook on C_2, they do not add more than z hops to its height.

The maximum height of a cluster is therefore $H_p = \max\{4z-5, z+\log_k |V|\}+z < 5z+\log_k |V|$. The time complexity of ζ is $4 \times H_p < 20z+4\log_k |V|$ per pulse. This is the same as the order of time complexity achieved by the z-partition algorithm.

5.3 A further improved z-partition algorithm

In this section we present a further improvement of the z-partition algorithm which reduces the communication complexity of ζ to $O(k|V|)$ — identical to the communication complexity of γ [1] and to the lower bound proved in [1]. This improvement makes the z-partition algorithm more complicated. Hence, we suggest that it would not be used in cases where $z \leq \sqrt{|V|}$, where the partition algorithm of Sec. 5.2 is satisfactory.

The main idea of this improvement is the following: after the improved z-partition protocol execution ends, an additional algorithm is executed. In this algorithm, every two neighboring z-clusters which contain less than z nodes are combined into one cluster. We refer to z-clusters with less than z nodes as *small z-clusters*, and other z-clusters as *large z-clusters*. The additional algorithm repeats the process of combining neighboring *small* z-clusters as long as possible. The detailed implementation of the further improved z-partition algorithm, as well as its time and communication complexities are discussed in [22]. The work [22] also contains a proof of its optimality.

The number of edges in the spanning forest is less than $|V|$. The number of preferred links between k-clusters remains $(k-1)|V|$ and the number of preferred links which connect z-clusters to k-clusters remains $|V|+z$.

The further improved partition algorithm ensures that there are no preferred links which connect small z-clusters. Therefore, when counting the number of

preferred links between z-clusters, it is sufficient to count only the links adjacent to large z-clusters.

There are at most $\lfloor \frac{|V|}{z} \rfloor$ large z-clusters, each of which may have at most z preferred links which connect it to other z-clusters. Hence, the number of preferred links connecting z-clusters is at most $\lfloor \frac{|V|}{z} \rfloor \times z \leq |V|$, and $E_p \leq |V| + (k-1)|V| + |V| + z + |V| = (k+2)|V| + z < (k+3)|V|$. Thus, when using the further improved partition algorithm, the communication complexity of ζ is $O(k|V|)$ per pulse. This is optimal and identical to the communication complexity achieved when using the original partition algorithm of [1].

Each new cluster created after the execution of the improved z-partition protocol, is composed of two clusters. Each of these clusters contains less than z nodes. Hence, the height of the composed cluster is at most $2z-3$, and the value of H_p remains $O(z + \log_k |V|)$. The time complexity of ζ remains, therefore, $O(z + \log_k |V|)$ per pulse.

6 Acknowledgments

The authors wish to thank Hagit Attiya and Shmuel Zaks for helpful discussions.

References

[1] B. Awerbuch, *Complexity Of Network Synchronization*, Journal of the Association for Computing Machinery **32**, 1985 804-823.

[2] B. Awerbuch, *Reducing Complexities of the Distributed Max-Flow and Breadth-First-Search Algorithms by Means of Network Synchronization*, Networks **15**, 1985 425-437.

[3] B. Awerbuch, *Optimal Distributed Algorithms of Minimum Weight Spanning Tree, Counting, Leader Election and Related Problems*, STOC 1987 230-240.

[4] M. Adam, Ph. Ingles and M. Raynal, *The Meaning of Synchronous Distributed Algorithms Run on Asynchronous Distributed Systems*, The Third International Symposium on Computer and Information Sciences, 1988 Izmir Turkey.

[5] B. Awerbuch and D. Peleg, *Network Synchronization with Polylogarithmic Overhead*, 31st Symp. on Foundations of Computer Science 1990 514-522.

[6] B. Awerbuch and M. Sipser, *Dynamic Networks are as fast as Static Networks*, Proc. 29-th IEEE symp. on Foundations of Computer Science 1988 206-220.

[7] C.T. Chou, I. Cidon, I.S. Gopal and S. Zaks, *Synchronizing Asynchronous Bounded Delay Networks*, IEEE Transactions on communications, **38**, 1990 144-147.

[8] S. Even and S. Rajsbaum, *Lack of Global Clock Does Not Slow Down the Computation in Distributed Networks*, Technical Report, TR-522, 1988, Computer Science Department, Technion IIT.

[9] S. Even and S. Rajsbaum, *The Use of a Synchronizer Yields Maximum Computation Rate in Distributed Networks*, Proc. 22-th ACM STOC, 1990 95-105.

[10] A. Fekete, N. Lynch and L. Shrira, *A Modular Proof of Correctness for a Network Synchronizer*, 2nd International Workshop on Distributed Algorithms, Amsterdam 1987.

[11] R. G. Gallager, P. A. Humblet and P. M. Spira, *A Distributed Algorithm for Minimum-Weight Spanning Trees*, ACM Transactions on Programming Languages and Systems **5**, 1983 66-77.

[12] E. Korach, G. Tel and S. Zaks, *Optimal Synchronization of ABD Networks*, Technical Report RUU-CS-88-23, 1988, Department of Computer Science, University of Utrecht, The Netherlands.

[13] K.B. Lakshmanan and K. Thulasiraman, *On The Use Of Synchronizers For Asynchronous Communication Networks*, 2nd International Workshop on Distributed Algorithms, Amsterdam 1987.

[14] K.B. Lakshmanan and K. Thulasiraman and M.A. Comeau, *An Efficient Distributed Protocol for Finding Shortest Paths in Networks with Negative Weights*, IEEE Transactions on Software Engineering **15**, 1989.

[15] Y. Malka and S. Rajsbaum, *Analysis of Distributed Algorithms based on Recurrence Relations*, 5th International Workshop on Distributed Algorithms, Delphi 1991.

[16] D. Peleg and J.D. Ullman, *An Optimal Synchronizer for the Hypercube*, SIAM Journal on computing **18**, 1989 740-747.

[17] S. Rajsbaum and M. Sidi, *On the Average of Synchronized Programs in Distributed Networks*, 4th International Workshop on Distributed Algorithms, 1990.

[18] A. Segall, *Distributed Network Protocols*, IEEE Transactions on Information Theory **IT-29**, 1983.

[19] L. Shabtay and A. Segall, *Active and Passive Synchronizers*, Technical Report, TR-706, 1991, Computer Science Department, Technion IIT.

[20] L. Shabtay and A. Segall, *Message Delaying Synchronizers*, 5th International Workshop on Distributed Algorithms, Delphi 1991.

[21] L. Shabtay and A. Segall, *On the Memory Overhead of Synchronizers*, LPCR Report #9313, 1993, Computer Science Department, Technion IIT.

[22] L. Shabtay and A. Segall, *A Version of the γ Synchronizer with Low Memory Overhead*, LPCR Report #9301, 1993, Computer Science Department, Technion IIT.