Distributed Shortest Paths Algorithms (Extended Abstract)

Baruch Awerbuch *
Dept. of Mathematics and Lab. for Computer Science,
M.I.T.,
Cambridge, MA 02139
ARPANET: baruch@theory.lcs.mit.edu

Abstract

This paper is concerned with distributed algorithm for finding shortest paths in an asynchronous communication network. For the problem of Breadth First Search, the best previously known algorithms required either $\Theta(V)$ time, or $\Theta(E + V \cdot D)$ communication. We present new algorithm, which requires $O(D^{1+\epsilon})$ time, and $O(E^{1+\epsilon})$ messages, for any $\epsilon > 0$. (Here, V is number of nodes, E is number of edges and D is the diameter.) This constitutes a major step towards achieving the lower bounds, which are $\Omega(E)$ communication and $\Omega(D)$ time.

For the general (weighted) shortest paths problem, previously known shortest-paths algorithms required $O(k \cdot V^2)$ messages and $O(V \cdot \log_k V)$ time. Our algorithm requires $O(E^{1+\epsilon} \cdot \log W)$ messages and $O(V^{1+\epsilon} \cdot \log W)$ time.

Our results enable to improve significantly solutions for other basic network problems (e.g. leader election).

1 Introduction

1.1 Model and complexity measures

This paper is concerned with distributed algorithms in an asynchronous communication network. Those algorithms may be repeated many times in case that the network's topology changes. From the point of view of network's performance, it is desirable that the messages of the control algorithms don't occupy much of the network bandwidth, and that such algorithms is performed relatively fast. Thus, we will be interested in minimizing the message exchange, as well as time of control algorithms. We deal exclusively with *worst-case* performance of the algorithm.

Given execution of a protocol in an asynchronous network, the *longest message delay* in that execution is the maximum difference between arrival and transmission times of a message, in terms of some global clock (which is not accessible to nodes). Following [Gal82], *normalized* time between two events in an execution is the ratio between the physical time between those two events in terms of the global clock above, and the longest message delay in that execution, i.e. physical time in case that link delays vary between 0 and 1. In context of asynchronous network, "time" always means "normalized time".

The *Communication Complexity* of π, denoted by c_π, is the upper bound on the number of messages sent in any execution of π. The *Time Complexity* of π, denoted by t_π, is the upper bound on on normalized time of any execution of π.

1.2 The problems

We consider the problem of finding shortest paths in the communication graph $G(V, E)$ of the network. In the "undirected" version of the problem, we are interested in finding shortest paths in an undirected graph $G_w = (V, E, w)$ from all nodes to an a distinguished nodes $s \in V$, where w is an assignment of *non-negative* weights w_e to edges $e \in E$.

The BFS problem is the special case of shortest paths, where $w(e) = 1$, for all $e \in E$. The "directed" version of the problem is defined similarly, except that we look for shortest paths in a directed graph $\bar{G} = (\bar{V}, \bar{E})$ which is an orientation of $G(V, E)$, i.e. $(i \rightarrow j) \in \bar{E}$ implies that $(i - j) \in E$.

Those problems appear to be fairly basic in the field of distributed network protocols. The major application for shortest paths tree is that shortest paths tree w.r.t. a given source node can be used for routing

*Supported by Air Force Contract TNDGAFOSR-86-0078, ARO contract DAAL03-86-K-0171, NSF contract CCR8611442, and a special grant from IBM.

of data from other nodes to the source node, in the cheapest possible way.

Improving efficiency of distributed Shortest Paths (BFS) yields improvement in more complex distributed algorithms, in which Shortest Paths (BFS) appears to be the bottleneck. Examples of such problems are given in section 2.

Observe that a shortest paths problem on network $G = (V, E, w)$ can be reduced to a BFS problem on an "expanded" network $\tilde{G} = (\tilde{V}, \tilde{E})$ where an edge e is substituted by a path containing w_e edges and $w_e - 1$ "dummy nodes". The difficulty with this reduction is that the number of nodes in the expanded network is huge, namely $|\tilde{V}| = O(W \cdot V)$, where W is the maximal edge weight. Using Gabow's scaling technique, one can effectively guarantee $W \leq |V|$. However, with Gabow's technique or without it, in either case $|\tilde{V}| \gg |V|$ and $|\tilde{E}| \gg |E|$. Thus, existence of a *black box* performing BFS with linear complexity, $O(\tilde{E})$ messages, only guarantees $O(E \cdot V)$ messages shortest paths algorithm, which is far from satisfactory.

2 Contributions of this paper

We present new distributed algorithms for the above problems with sharply improved bounds on communication and time.

Our algorithm is based on novel synchronization technique, which proceeds recursively. Even though the algorithm is just a composition of very simple modular blocks, its analysis is non-trivial. To overcome the difficulties, we introduce novel counting methods, based on notions of *amortized* complexities.

2.1 Improvements for BFS

The obvious lower bounds on communication and time complexities of distributed BFS algorithms are $\Omega(E)$ messages and $\Omega(D)$ time, where E is the number of network edges, and D is the diameter. In the *synchronous* network, the obvious algorithm meets those lower bounds. However, the situation is much more complex in the *asynchronous* network, where the best known algorithms exhibit *trade-off* in terms of *communication* and *time* complexities.

The best known algorithm in terms of *communication* is due to [AG85]. It requires $\Theta(E^{1+\epsilon})$ messages and $\Theta(V^{1+\epsilon})$ time, for all $\epsilon > 0$. [AG85] is "relatively close" to the lower bound in communication. However, as advocated by Peleg [Pel87], this algorithm is quite inefficient in time, in case that $D \ll V$. Peleg [Pel87] points out that the difference between $O(V)$ and $O(D)$ time can be very significant in many existing networks,

Author	Communication	Time
[Awe85]	$E + V \cdot k \cdot D$	$D \cdot \log_k V$
[AG85]	$E^{1+\epsilon}$	$V^{1+\epsilon}$
This paper	$E^{1+\epsilon}$	$D^{1+\epsilon}$

Figure 1: Our BFS algorithms versus existing ones.

Author	Communication	Time
[Awe85]	$k \cdot V^2$	$V \cdot \log_k V$
[Fre85]	$V^{1\frac{7}{8}}$	$V^{1\frac{7}{8}}$
This paper	$E^{1+\epsilon} \cdot \log W$	$V^{1+\epsilon} \cdot \log W$

Figure 2: Our Shortest Paths algorithms versus existing ones.

e.g. the ARPANET [MRR80], where $D \ll V$. The high ($\Omega(V)$) time overhead is inherent for that method.

The best known algorithm in terms of *time* is achieved by applying *synchronizer* γ of [Awe85] to the obvious synchronous algorithm. This algorithm (symbolically attributed to [Awe85]) requires $O(E+V \cdot D \cdot k)$ messages and $O(\log_k V \cdot D)$ time, for any k. It meets the lower bound in time but is quite inefficient in communication.

The new BFS algorithm requires $O(E^{1+\epsilon})$ messages and $O(D^{1+\epsilon})$ time, for all $\epsilon > 0$, thus making a step towards achieving the lower bounds. The following Figure 1 summarizes results of this paper, comparing them to existing results.

2.2 Improvements for Shortest Paths

There have been a number of works on shortest paths in the past [Gal82], [Jaf80]. The best previously known shortest-paths algorithms is obtained using the SYNCHRONIZER of to [Awe85]. It requires $O(k \cdot V^2)$ messages and $O(V \cdot \log_k V)$ time. (This does not count the preprocessing phase of [Awe85].) For *planar* graphs only, the algorithm of Frederickson achieves $O(V^{1\frac{7}{8}})$ message and time.

Using the scaling techniques of Gabow [Gab85] and the BFS algorithm in this paper, we obtain a new shortest paths algorithm, which requires $O(E^{1+\epsilon} \cdot \log W)$ messages and $O(V^{1+\epsilon} \cdot \log W)$ time.

The following Figure 2 summarizes our improvements over existing algorithms.

2.3 Applications for other problems

Leader Election, Spanning Tree, Global Functions: One of the most well-studied problems in the field of distributed network algorithms is the problem

Author	Communication	Time
[Awe87]	$E + V \cdot \log V$	V
[Pel87]	$E \cdot D$	D
This paper	$E^{1+\epsilon}$	$D^{1+\epsilon}$

Figure 3: Our Leader Election algorithms versus existing ones.

of finding a leader in a network. It is equivalent to the problem of finding a spanning tree. Leader election is an important tool for breaking symmetry in a distributed system. Construction of a spanning tree or finding a leader appears as a building block essentially in every complex network protocol, and is closely related to many problems in distributed computing. There are many problems which are provably equivalent [Awe87] to the problem of electing a leader, in terms of the communication and time complexities. One example of such problems is a class of so-called "global sensitive" functions [Awe87], e.g. MAXIMUM, SUM, PARITY, MAJORITY, COUNTING, OR, AND.

The best known leader election algorithms have been given in [Awe87], [Pel87]. Peleg [Pel87] advocates for leader election algorithms whose running time depends only on the diameter of the network. While Peleg's algorithms achieves optimal $O(D)$ time, its communication complexity is $O(E \cdot D)$.

Using the new BFS algorithm in this paper, we can improve significantly time performance of existing leader election algorithms. We achieve almost linear (in diameter) time and almost linear communication. Our improvements are shown in following Figure 3.

Compact Routing Tables: For the purpose of constructing compact routing tables, the best current algorithm due to Peleg and Upfal [PU88] uses network partition algorithms, which are modifications of the *synchronizer-initialization* algorithm of [Awe85]. BFS is the bottleneck in this algorithm.

2.4 Organization of this paper

In this extended abstract, we only deal with BFS algorithms. We leave the extensions to the Shortest Paths, Leader Election, etc., for the full paper. Although our BFS algorithms is in fact very simple, it has a recursive structure, which makes it hard to conceive it as a whole.

In the following Section 3 we describe the basic tools used. In Section 4 we outline main ideas behind our improvement, and provide more details in Sections 5, 6, 7, 8, 9. In Section 10 we analyze complexity of the algorithm.

3 Basics

3.1 DIJKSTRA algorithm

Let us first outline a simple BFS algorithm, which will be (symbolically) referred to as the DIJKSTRA Algorithm, because of its similarity with Dijkstra's shortest path algorithm and Dijkstra-Sholten distributed termination detection procedure [DS80].

The algorithm maintains a tree rooted at the source node. Initially, the tree is empty. Upon termination of the algorithm, the tree is the desired BFS tree. Thruout the algorithm, the tree can only grow, and at any time it is a sub-tree of the final BFS tree. The algorithm operates in successive iterations, each processing another BFS layer. At the beginning of a given iteration l, the tree has been constructed for all nodes in layers $m < l$. Upon the termination of iteration l, the tree will be extended by one layer, covering also all nodes in layer l.

The purpose of the source node is to control these iterations by means of a synchronization process, performed over the tree. This enables the source node to detect the time that the tree has been completed up to distance l and thus iteration $l + 1$ can be started.

The source node triggers each iteration by broadcasting message over the tree which is forwarded out to nodes at layer $l - 1$ in the tree. Upon receipt of this message, the latter nodes send "exploration" messages to all neighbors, carrying label $l - 1$, and trying to discover nodes at layer l.

When a node receives exploration message from a neighbor, it acts as follows. If this is the first exploration message that the node has seen, it chooses the sender of the message as its parent, sets its distance label to be 1 plus distance label of the sender, and sends back a "positive" acknowledgment (ack) to the sender, indicating that the sender was chosen as a parent. Upon receipt of subsequent exploration messages, the node sends back to sender "negative" ack, indicating that it already has parent. Upon receiving a positive ack, a node adds the sender to its list of children.

When a node at layer $l - 1$ receives acks to all exploration messages it has sent, it sends an ack to its parent in the BFS forest, indicating whether any new descendants have been discovered. When an internal node gets such acks from all children, it sends an ack to its parent. Eventually, all the acks are collected by the source node. This implies that layer l has been processed completely. If any nodes have been discovered at that layer, the next iteration $l + 1$ is started. Otherwise, the algorithm terminates.

The complexities of this algorithm are $O(V \cdot D + E)$

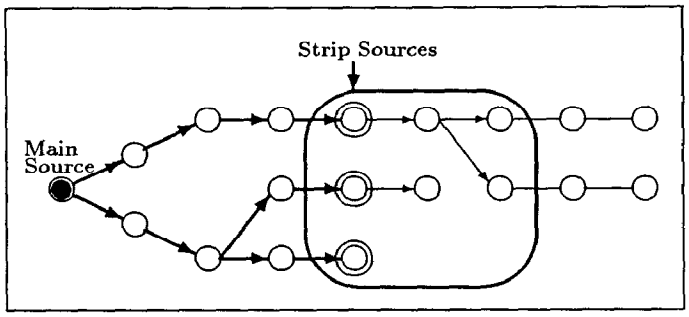

Figure 4: Strip Method.

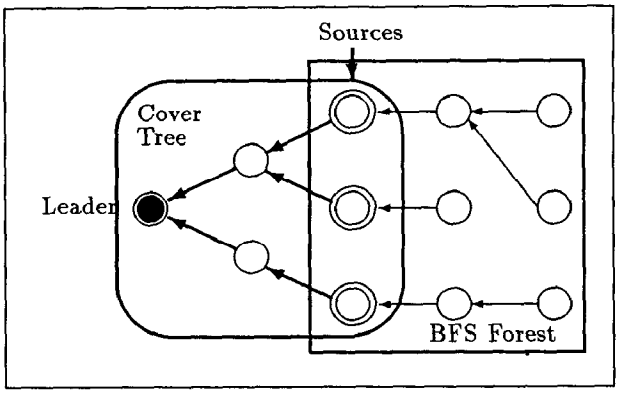

Figure 5: Data structures associated with a cluster.

messages and $O(D^2)$ time, where d is the number of layer being processed, Indeed, there are D iterations and in each of them synchronization is performed over BFS tree which requires $O(V)$ messages and $O(D)$ time. In addition, one exploration message is sent over each edge once in each direction.

The overhead due to synchronization makes this algorithm quite inefficient in sparse and long networks, where $E \ll V \cdot D$. If one considers a network with all nodes on a single path of length $V-1$, one sees that the communication and time complexity are each $O(V^2)$. Obviously, the performance of the algorithm degrades as the number D of layers to be processed increases.

3.2 Distance reduction paradigm

It is easy to reduce the problem where big number of layers needs to be processed to problem where small number of layers needs to be processed. If the network has diameter D, we can conceptually "cut" the network into $\frac{D}{d}$ "strips" of length d, and process those strips sequentially, like in the following Figure 4.

Our strategy now is to cut the network into strips of size $d < D$, and process those strips one after another, thus extending the BFS tree. We know also which edges lead to nodes in previous strip, so that messages are not sent along those edges.

For the purpose of processing the strip, we need to create BFS forest rooted at the all nodes on the border of the strip. Thus, we have *multiple* source nodes, rather than single source node.

The most naive reduction from the case of multiple sources to the case of single source is to find *separately* shortest paths w.r.t. each one of sources, and then "combine" those shortest paths in an obvious manner. However, this strategy is of order of magnitude of the number of nodes in the cluster. Unfortunately, this method may introduce a blow-up factor to the communication complexity, which grows with the number of layers in the strip and the number of sources in the strip. Using this technique, with strips of size $d = \sqrt{D}$, we can achieve $O(D^{1.5})$ time and $O(E \cdot \sqrt{D})$ communication. This strategy, with some additional improvements, has been used in [AG87], and in [Fre85].

We are looking for more efficient reduction strategy. In sequential setting, one can easily reduce the problem with multiple sources to the problem with a single source, by connecting each one of the sources to some auxiliary ("root") source node via edges of weight 0, or simply contract all the sources into a single source. In a distributed setting, this method does not work.

3.3 Source contraction paradigm

However, whenever we encounter the a multiple-sources problem, we will always have available some "cover tree", spanning all those nodes. For example, there is a legitimate cover tree spanning all the source nodes of a strip, namely the (whole) BFS tree spanning all the previous strips. The cover tree can be used to synchronize between source nodes, thus effectively "contracting" them into a single (super-)node, or *cluster*. With each cluster, we associate the following distributed data structures:

- *Source nodes* of the cluster.
- *Cover tree* which spans all the source nodes.
- *BFS forest* which spans all the sources.

We can use those data structures in order to run DIJKSTRA algorithm for the case of multiple sources as if it were run in the case of single source. The basic idea is that the algorithm proceeds in iterations, controlled by the "root". The synchronization over cover tree enables the root to detect that *all* the sources have completed their trees up to a certain BFS layer. The

root node notifies all the sources about the beginning of the iteration by broadcasting message over the cover tree. Upon receiving this messages, each source node node runs one iteration of DIJKSTRA algorithm, as described above. When a source node collects all its acks to messages over BFS tree rooted at itself, its task is complete. The source node will send ack towards the root of the cover tree after it has completed its task, and after it receives acks from all its children. Nodes propagate acks in this manner, until the root node has received acks to all the messages that it has sent. Now, new iteration can start.

Clearly, the communication overhead of synchronization will grow with the *size* of the cover tree, while time overhead of synchronization will grow with the *depth* of the cover tree, i.e. the length of a longest path from a node to the root. We conclude that in order for the resulting algorithm is efficient, the size of cover tree should not be much bigger than the the strip being processed, both in terms of the size and depth.

Observe that, initially, we have a cover tree for the strip, which is the whole BFS tree. It has depth of $\Theta(D)$ and size of $\Theta(V)$, i.e. is way too big and too long. The natural strategy is to reduce hard problems to easy problems. In order to do this, we have to reduce the original problem with "big" cover tree and "big" processing length, to "not too many" new problems with "big" cover tree and "big" processing length.

As noticed in [AG85], we can treat separately different connected components in the strip, since trees grown in different components do not interfere with each other. Thus, the algorithm of [AG85] was trying to construct a spanning tree of each connected component of the strip. The difficulty here is that the set of nodes belonging to the strip is not known in advance; we know where the strip "starts", but we do not know where it "ends". If one knows in advance where the strip "ends", the problem solved in [AG85] is trivialized.

An algorithm proposed in [AG85] enables to construct recursively spanning trees of each strip. This strategy guarantees that cover tree have small number of nodes, but, unfortunately, tend to have very big depth. The reason for this is that a connected component of a strip of depth d does not necessarily have a spanning tree of depth d. In fact, it might be the case that the whole strip is connected, and that any tree that will span all the source nodes of the strip will have $\Omega(V)$ depth, causing $\Omega(V)$ time overhead. Since this method inherently requires $\Omega(V)$ time overhead, it is inadequate for us. It is worth pointing out that [AG85] focused only on communication.

The main contribution of this paper is the reduction of the problem with big cover tree to "moderate" number of problems with cover trees of "moderate" depth and size.

3.4 Strip Cover

Definition 3.1 Given a strip with d BFS layers, a *strip cover* is a forest of node-disjoint trees, which span all the source nodes in the strip. A collection of clusters *induced* by the cover is a set of subsets of source nodes, each subset consisting of the set of all source nodes spanned by the same tree of the cover.

Definition 3.2 For an arbitrary strip cover (forest) define the following parameters:

load factor: is the maximal number of clusters which are within distance d from some node in the strip.

depth factor: is the maximal depth of a tree in the cover, *divided* by d.

size: is the number of trees in the cover.

Our task would have been significantly simplified, if, *prior* to processing this strip, some "oracle" would give us a "good" strip cover, for which both *load factor* and *depth factor* being "small". We can then process the strip "efficiently" by contracting all the sources spanned by the same tree into a single cluster, and then performing BFS independently from each cluster. The intuition here is that *load factor* is the reason for communication blow-up, as it upper-bounds the number of different clusters competing for the same node. Also, *depth factor* is the reason for time blow-up, as it upper-bounds the size of the trees on which synchronization is performed.

By "refining" the cover, i.e. increasing its *size*, we reduce *depth factor* on expense of increasing *load factor*. For example, the "coarsest" cover (all sources in the same tree) may feature *depth* $= V$ and *load* $= 0$. On the other extreme, the "finest" cover (all sources in different trees) may feature *depth* $= 0$ and *load* $= V$. It is easy to see that we can achieve the following compromise.

Fact 3.3 There always exists a cover with both *load factor* and *depth factor* being at most $V^{\frac{1}{\sqrt{\log_2 V}}}$.

It is not obvious how Fact 3.3 will benefit us, since constructing such "good" cover appears to be as hard as performing the BFS itself. This difficulty is resolved by running, in parallel, approximations for BFS and for "good" cover, as shown in the following sub-section.

4 Outline of our algorithm

Our main contribution is a novel algorithm, referred to as STRIP-BFS, which processes a strip with d layers. We "pretend" to perform BFS independently from each source node. Since $d \ll D$, then (recursively) we assume existence of efficient BFS algorithm that processes distance d w.r.t. single node. However, to save communication, we require that after a given node v enters into X BFS trees, where X is a parameter. After that, it will not enter any additional trees, causing those trees to become *blocked*. This algorithm is called MULTI-BFS with parameter X. Observe that naive strategy of performing BFS's independently from each source node corresponds to MULTI-BFS with the choice of $X = \infty$. The approach used in [AG85] corresponds to MULTI-BFS with $X = 1$.

Clearly, trees that are blocked will not extend to the required length. As a result, some nodes will appear in "wrong" trees, and we will not be able to reconstruct a real shortest-path forest from the collection of individual BFS trees (as in case $X = \infty$). However, the information obtained will help us to construct short paths between source nodes, that can be used later for synchronization between those nodes. Namely, to correct the situation, we "contract" blocked source nodes into *clusters* of nodes, each cluster having a relatively short cover tree. This tree is obtained by "stitching" together BFS trees of the involved source nodes.

Our algorithm proceeds to find true BFS forest of a strip in a number of iterations. The major data structures maintained by the the algorithm is the cover of the strip, a collection of clusters, induced by that cover. Initially, each cluster is a (singleton) source, and all cover trees are degenerate trees each containing single source.

Each iteration consists of two phases, "BFS" phase, and "Merge" phase. In the "BFS" phase, all clusters run MULTI-BFS algorithm. Upon termination of "BFS" phase, we examine the resulting collection of BFS trees, and record in memory all unblocked trees. In the "Merge" phase, all clusters whose BFS trees got blocked, get merged into bigger clusters. If none are left, the main loop of the algorithm terminates. This must happen eventually since once clusters will become big enough, there will not be enough clusters to cause blocking. At this time, we invoke the EXTRACT procedure, that extracts the BFS forest of the strip from the collections of all BFS forests, that were recorded as unblocked during some iteration.

In order to maintain small communication complexity, we need to guarantee that number of trees passing thru a node is small, which suggests that X should be as small as possible. Also, in order to maintain small time and communication complexity, the number of iterations should be small, suggesting that "Merge" phase tries to merge as many clusters as possible. Unfortunately, this policy causes the cover trees of the resulting cover to become very big, much bigger than d, namely $\Theta(V)$. This immediately leads time complexity to skyrocket up to $\Omega(V)$, as in [AG85]. (The number of iterations in [AG85] is only $\log_2 V$.)

We present an algorithm, referred to as MERGE algorithm, whose main idea is to try to combine together as many clusters as possible, subject to the restriction that only the clusters which are close to each other, namely within distance $O(d)$, can be combined together.

The crucial parameters for performance of the algorithm are *depth factor* and the *size* of the cover. The effect of one application of MERGE algorithm on the cover is that

- *size* is reduced by *at least* factor of X.
- *depth* is increased by at *at most* factor of 5.

This guarantees that the number of iterations is at most z, and that all cover trees have small depth, namely of depth $O(d \cdot 5^z)$, where $z = \log_X V$.

The algorithm uses a variation of the *deterministic* symmetry breaking technique of [AGLP88]. This is in turn a variation of the Luby's maximal independent set algorithm [Lub86] and Luby's technique for removing randomness from distributed computing [Lub88]. This enables to break the symmetry in the network in $O(d \log V)$ expected time, where d is the depth of a spanning tree.

Overall, we accomplish a reduction from the problem of processing strips of size D to the problem of processing strips of size $d < D$. We can continue *recursively*, to reduce the size of the strip to be processed, until we end up with a strip containing a single layer, at which point it does not matter which algorithm is used.

The schematic description of all the subroutines is given in the following Figure 6. The "main" BFS algorithm is referred to as MAIN-BFS. It calls, as a subroutine, STRIP-BFS which processes strips of smaller size, by calling MULTI-BFS, MERGE, and EXTRACT. Finally, MULTI-BFS calls MAIN-BFS. This illustrates the fact that our algorithm is recursive.

In the following sections 5, 6, 7, 8, 9 we describe MAIN-BFS, STRIP-BFS, MULTI-BFS, MERGE, and EXTRACT algorithms, respectively.

5 MAIN-BFS algorithm

Definition 5.1 (Input/Output) *Input* of MAIN-BFS consists of

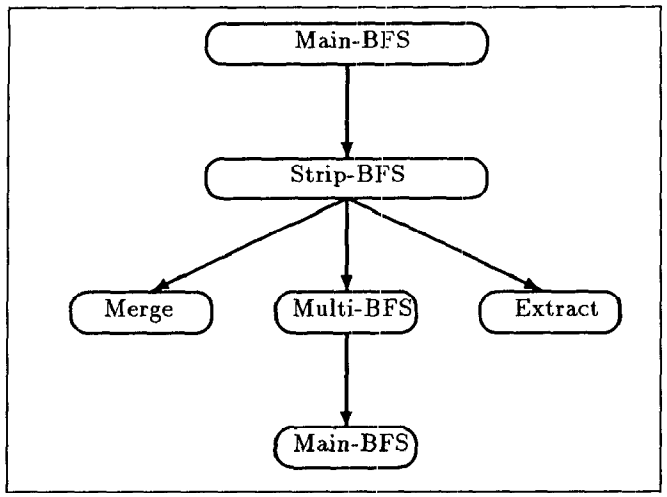

Figure 6: General Structure.

Variables

Main-Source: the sources of the strip. (*Input* variable.)

Main-BFS: the BFS w.r.t. the source node of the process. (*Output* variable.)

Main-depth: number of layers to be processed. (*Input* variable.)

Strip-depth: number of layers in a strip. (*Input* variable.)

♯Strip: the number of the strip being processed.

Sources: sources of a strip.

Strip-BFS: the BFS forest w.r.t. sources of a strip.

Procedures

STRIP-BFS: constructs BFS forest of a strip w.r.t. **Sources**.

Figure 7: Declarations of the MAIN-BFS algorithm.

Main-Source: the *single* source node.

Main-depth: the number of layers to be processed by the MAIN-BFS algorithm.

Strip-depth: is the number of layers in a strip.

Output of MAIN-BFS consists of the tree **Main-BFS**, grown out of **Main-Source**.

The algorithm itself is a straightforward implementation of the "Distance reduction paradigm" of Section 3.2. It partitions the network into strips of smaller size and processes strips one by one, employing a procedure STRIP-BFS which extends BFS forest for a another strip of a length **Strip-depth**.

Thruout the algorithm, **Sources** denotes the sources of the current strip (which are the "frontier" of the existing BFS tree) and **♯Strip** denotes the number of current strip. Once BFS tree cannot expand any more, i.e. is stuck in a "dead end" (in which case **Sources** $= \emptyset$), or all the strips have been processed, (in which case **♯Strip** $= \frac{\textbf{Main-depth}}{\textbf{Strip-depth}}$), the algorithm terminates.

The declarations and the code of the algorithm are presented in Figures 7, and 8, respectively.

```
♯Strip ← 1
Sources ← Main-Source
Main-BFS ← ∅
repeat
   Strip-BFS ← STRIP-BFS
   add Strip-BFS to Main-BFS
   ♯Strip ← ♯Strip + 1
   Sources ← frontier of Main-BFS
until ♯Strip = Main-depth / Strip-depth or Sources = ∅
return Main-BFS
```

Figure 8: Algorithm MAIN-BFS.

6 STRIP-BFS algorithm

Definition 6.1 (Input/Output) *Input* of STRIP-BFS consists of

Sources: the set of source nodes of the strip.

Main-BFS: the existing BFS tree.

Output of MAIN-BFS consists of **BFS**, which is the BFS forest of strip, grown out of **Sources**.

The algorithm proceeds in phases, so that at a given time all the nodes execute the same phase. The tree **Main-BFS** is used in order to detect termination of the previous phase and to trigger the next one.

Thruout the algorithm, $\mathcal{V}$ denotes the current set of clusters. After application of MULTI-BFS, the set $\mathcal{U}$ contains all clusters $s \in \mathcal{V}$ whose BFS forest is not blocked, and the complementary set $\mathcal{B} = \mathcal{V} \backslash \mathcal{U}$ contains the blocked clusters. Next, unblocked clusters $s \in \mathcal{U}$ join the set $\mathcal{A}$ of all clusters that were unblocked in the past, and thus (implicitly) forests **Mbfs**$_s$ join the collection $\mathcal{MBFS}_{\mathcal{A}}$ of all unblocked BFS forests.

If no unblocked clusters remain, i.e. $\mathcal{B} = \emptyset$, then the following application of MERGE returns $\mathcal{V} = \mathcal{COVER}_{\mathcal{V}} = \mathcal{SOURCES}_{\mathcal{V}} = \emptyset$ and the algorithm terminates. Otherwise, MERGE merges all remain-

Procedures

MULTI-BFS: Executes procedure MAIN BFS "in parallel" at each cluster, allowing node to enter at most X forests.

MERGE: Merges together all blocked clusters.

EXTRACT: Extracts the "best" BFS forest from collection of trees.

Variables

Sources: the sources of the strip. (Input variable).

BFS: The final BFS forest of the strip. (Output variable).

$\mathcal{V}$: all clusters in current iteration.

$\mathcal{U}$: clusters that became unblocked in *last* iteration.

$\mathcal{B}$: clusters that became blocked in *last* iteration.

$\mathcal{A}$: clusters that became unblocked in *all* previous iterations.

$\mathcal{SOURCES}_{\mathcal{V}}$: collection of source sets of all clusters.

$\mathcal{MBFS}_{\mathcal{V}}$: collection of BFS forests of all current clusters.

$\mathcal{MBFS}_{\mathcal{A}}$: collection of *all* unblocked BFS forests.

$\mathcal{COVER}_{\mathcal{V}}$: collection of cover trees of all current clusters.

Figure 9: Declarations of the STRIP-BFS algorithm.

```
V ← {s|s ∈ Sources}                          /* initial clusters */
A ← ∅                                        /* no unblocked clusters */
SOURCES_V ← {s|s ∈ Sources}                  /* initial sources */
COVER_V ← ∅                                  /* initial cover is empty */
MBFS_A ← ∅                                   /* no unblocked BFS forests */
repeat                                       /* loop which processes a given strip */
   MBFS_V ← MULTI-BFS                        /* grow BFS forests */
   U ← {s|s ∈ V, s unblocked in MBFS_V}
   B ← V \ U                                 /* the rest are blocked clusters */
   A ← A ⋃ U                                 /* remember unblocked BFS forests */
   (V, COVER_V, SOURCES_V) ← MERGE
until V = ∅                                  /* no blocked clusters sources left */
BFS ← EXTRACT(MBFS_A)                        /* extract true BFS forest */
return BFS
```

Figure 10: Algorithm STRIP-BFS.

ing blocked clusters $s \in \mathcal{B}$ into bigger clusters. Now $\mathcal{V}$, $\mathcal{COVER}_{\mathcal{V}}$, $\mathcal{SOURCES}_{\mathcal{V}}$ denote, respectively, set of new clusters, new cover, and new sources. At this point, MULTI-BFS is called again.

The declarations and the code of the algorithm are presented in Figures 9, and 10, respectively.

7 MULTI-BFS algorithm

7.1 Specifications of MULTI-BFS

Notation 7.1 We denote by $v \in \mathcal{MBFS}_{\mathcal{V}}$ the fact that a node v is spanned by one of the forests of the collection $\mathcal{MBFS}_{\mathcal{V}} = \{\texttt{Mbfs}_s | s \in \mathcal{V}\}$. For each node $v \in V$, we define

- $Load_v(\mathcal{MBFS}_{\mathcal{V}})$ is the number of forests in $\mathcal{MBFS}_{\mathcal{V}}$ which span v. (Is 0 if $v \notin \mathcal{MBFS}_{\mathcal{V}}$.)
- $Distance_v(\texttt{Mbfs}_s)$ is the distance between node v and the root of a tree in $\texttt{Mbfs}_s$. In case that $v \notin \texttt{MBFS}_s$, $Distance_v(\texttt{Mbfs}_s) = \infty$.

Definition 7.2 (Input/Output) *Input* to MULTI-BFS consists of

$\mathcal{V}$: a set of indices.

$\mathcal{COVER}_{\mathcal{V}} = \{\texttt{Cover}_s | s \in \mathcal{V}\}$: a cover of $\mathcal{SOURCES}_{\mathcal{V}}$.

$\mathcal{SOURCES}_{\mathcal{V}} = \{\texttt{Sources}_s | s \in \mathcal{V}\}$: clusters induced by the cover.

X: collision threshold.

d: distance threshold.

Output of MULTI-BFS is a collection of forests $\mathcal{MBFS}_{\mathcal{V}} = \{\texttt{Mbfs}_s | s \in \mathcal{V}\}$ such that

- $\texttt{Mbfs}_s$ is a BFS forest w.r.t. $\texttt{Sources}_s$ in the subgraph induced by the nodes of $\texttt{Mbfs}_s$.
- For any $v \in V$, $Load_v(\mathcal{MBFS}_{\mathcal{V}}) \leq X$.
- For any $v \in \texttt{Mbfs}_s$, $Distance_v(\texttt{Mbfs}_s) \leq d$.
- For any $v \in \texttt{Mbfs}_s$, and any edge $(v, u) \in E$, *at least* one of the following conditions must hold:
  - $u \in \texttt{Mbfs}_s$.
  - $Distance_v(\texttt{Mbfs}_s) = d$.
  - $Load_u(\mathcal{MBFS}_{\mathcal{V}}) = X$.

Intuitively, $\mathcal{MBFS}_{\mathcal{V}}$ is a collection of BFS trees, grown "independently" from each one of the sources $s \in S$ for d layers, under the constraint a node may belong to at most X such trees, and depth of each tree is at most d. That is, node that already belongs to X trees will not enter additional trees, and nodes at depth d cannot have any children. Thus, for each edge $(v \rightarrow u)$ outgoing from a node $v \in \texttt{Mbfs}_s$, either $u \in \texttt{Mbfs}_s$, namely node u is in the same forest, or u has not been included because of one of the two reasons. First reason is that $Distance_v(\texttt{Mbfs}_s) = d$, i.e. v is at the "last layer" and thus forest cannot grow from v any more. Second reason is that $Load_u(\mathcal{MBFS}_{\mathcal{V}}) = X$, i.e. u is "overloaded" and refuses to enter the forest.

7.2 Implementation of MULTI-BFS

Each cluster runs MAIN-BFS algorithm for the required length. To simplify programming of the algorithm, we initially allow nodes to enter more than X forests. However, a node may *grow* at most X forests, and *blocks* additional BFS processes by refusing to grow their forests. At the end of the algorithm we delete the node from all those forests, so that, ultimately, each node belongs to at most X forests.

Towards that goal, messages of the process MAIN-BFS$_s$, which grows BFS forest from a cluster s, are tagged with parameter s. Each node stores in `List` identities of all clusters whose forests node belongs to, excluding forests blocked by the node. When a node receives a message of MAIN-BFS$_s$, it will "refuse" to grow this forest in case that $|\texttt{List}| = X$ and $s \notin \texttt{List}$, i.e. it has already grown forests of X clusters other than s.

We achieve the effect of blocking a process by only modifying the local input of that process. Namely, node pretends that the edge on which the message of the process has arrived is the *only* edge adjacent to that node, thus effectively disabling growth of that forest thru the node.

This is implemented as follows. Node maintains variable `Edges`, containing the local topology as seen by the MAIN-BFS processes running at the node, and variable `All-Edges`, which contains *true* local topology. Node will distinguish between MAIN-BFS processes which are blocked by the node and those which are not, by setting appropriately `Edges` variable prior to responding to message of the process. For processes that are *not* blocked, node will set `Edges` := `All-Edges`, i.e. use "true" local topology. For processes that *are* blocked, node will set `Edges` $= e$, i.e. use "fake" local topology, consisting of the single edge e, on which message has arrived.

The declarations and the code of the algorithm are presented in Figures 11 and 12, respectively.

Procedures

MAIN-BFS$_s$: recursive call to MAIN BFS from cluster l.

Variables

`All-Edges`: the local topology, i.e. set of incident links.

`Edges`: the local topology, i.e. set of incident links, as seen by the BFS algorithm. Initially, `Edges` = `All-Edges`.

`List`: the list of all clusters, whose BFS forests currently pass thru the cluster.

Figure 11: Declarations of the MULTI-BFS algorithm.

```
Message of MAIN-BFS_s over edge e
  Edges ← All-Edges          /* default is true topology */
  if s ∉ List then                    /* new BFS process */
    if Counter < X then               /* no need to block */
      List ← List ⋃ s          /* participate in s's forest */
      Counter ← Counter + 1      /* increment counter */
    else Edges ← {e}           /* block this BFS forest */
  invoke MAIN-BFS_s   /* view Edges as local topology */
```

Figure 12: Algorithm MULTI-BFS.

8 MERGE algorithm

8.1 Specifications of MERGE

Definition 8.1 (Input/Output) *Input* of MERGE consists of

$\mathcal{V}$: a set of indices.

$\mathcal{B}$: a subset of $\mathcal{V}$, containing all indices whose BFS forests have been blocked.

$\mathcal{SOURCES}_{\mathcal{V}} = \{\texttt{Sources}_s | s \in \mathcal{V}\}$: a collection of node-disjoint clusters $\texttt{Sources}_i \subset V$.

$\mathcal{COVER}_{\mathcal{V}} = \{\texttt{Cover}_s | s \in \mathcal{V}\}$: a forest of node-disjoint trees such that $\texttt{Cover}_i$ spans $\texttt{Sources}_i$.

$\mathcal{MBFS}_{\mathcal{V}} = \{\texttt{Mbfs}_s | s \in \mathcal{V}\}$: a collection of node-disjoint forests of depth d at most such that $\texttt{Mbfs}_i$ is forest rooted at the nodes of $\texttt{Sources}_i$.

Output of MERGE consists of

$\overline{\mathcal{V}}$: a new set of indices.

$\overline{\mathcal{SOURCES}}_{\overline{\mathcal{V}}} = \{\overline{\texttt{Sources}}_p | p \in \overline{\mathcal{V}}\}$: new clusters.

$\overline{\mathcal{COVER}}_{\overline{\mathcal{V}}} = \{\overline{\texttt{Cover}}_p | p \in \overline{\mathcal{V}}\}$: new cover.

Notation 8.2 For a cover $\mathcal{C}$, we denote by $Depth(\mathcal{C})$, $Load(\mathcal{C})$, $Size(\mathcal{C})$ the depth factor, the load factor, and the size of $\mathcal{C}$, respectively.

Definition 8.3 The output of MERGE must satisfy

$$\bigcup\{\overline{\texttt{Sources}}_p | p \in \overline{\mathcal{V}}\} = \bigcup\{\texttt{Sources}_s | s \in \mathcal{B}\} \quad (1)$$

$$Depth(\overline{\mathcal{COVER}}_{\overline{\mathcal{V}}}) \leq 5 \cdot Depth(\mathcal{COVER}_{\mathcal{V}}) \quad (2)$$

$$Size(\overline{\mathcal{COVER}}_{\overline{\mathcal{V}}}) \leq \frac{Size(\mathcal{COVER}_{\mathcal{V}})}{X} \quad (3)$$

(1) means that the new sources consist of all sources belonging to clusters blocked in previous iteration. (2) means that the depth of new clusters increases by factor of 5 at most. (3) means that the number of new clusters decreases by factor of X at least.

8.2 Implications for STRIP-BFS

Observe that at each application of MERGE, except, perhaps, for the last one,

$$Size(\mathcal{COVER}_{\mathcal{V}}) \geq Load(\mathcal{COVER}_{\mathcal{V}}) \geq X \qquad (4)$$

since otherwise, no blocking occurs and the algorithm terminates. Since in the first iteration, $Size(\mathcal{COVER}_{\mathcal{V}}) \leq V$, we deduce that, in STRIP-BFS,

Corollary 8.4 There are at most $\log_X V$ iterations.

Since, at the first iteration, $Depth(\mathcal{COVER}_{\mathcal{V}}) = 1$, we deduce that, at *any* iteration,

$$Depth(\mathcal{COVER}_{\mathcal{V}}) \leq d \cdot 5^z \leq d \cdot V^{\frac{1}{\sqrt{\log_5 V}}} \qquad (5)$$

Thus, thruout all the iterations of STRIP-BFS, the cover used is quite "shallow". This is the key fact for upper-bounding the time complexity.

8.3 Implementation of MERGE

Notation 8.5 Let the *collision graph* $\mathcal{G}(\mathcal{V}, \mathcal{E})$ be undirected graph where

$$\mathcal{E} = \{(r,s) | r,s \in \mathcal{V}, \exists v \in \texttt{Mbfs}_s, \exists u \in \texttt{Mbfs}_r, (u,v) \in E\}$$

Namely, this is undirected graph, whose nodes represent clusters, and edges represent adjacency of `Mbfs` forests.

Notation 8.6 Let $degree_{\mathcal{G}}(v)$ and $distance_{\mathcal{G}}(u,v)$ denote the degree of v and the distance between u and v in $\mathcal{G}$.

Definition 8.7 Given an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a subset $\mathcal{B} \subset \mathcal{V}$ of of *blocked* vertices, we say that a subset $\overline{\mathcal{V}} \subset \mathcal{V}$ of vertices is a *Small Ruling Sub-Set (SRS)* of $\mathcal{B}$ in $\mathcal{G}$ if the following conditions hold:

$$\forall u \in \mathcal{B}, \exists v \in \overline{\mathcal{V}} \text{ such that } distance_{\mathcal{G}}(u,v) \leq 2 \qquad (6)$$

$$|\overline{\mathcal{V}}| \leq \frac{|\mathcal{V}| \log |\mathcal{V}|}{d} \text{ where } d = \min_{v \in \mathcal{B}} \{degree_{\mathcal{G}}(v)\} \qquad (7)$$

For every forest that is blocked, there exists *at least* one edge leading from a node in the forest to a node blocking that forest, which participates in *at least* X other forests. Thus, with our particular choice for $\mathcal{G}$ and $\mathcal{B}$, we have $d \geq X \geq |V|^{\frac{1}{\sqrt{\log_5 |V|}}}$.

The algorithm itself proceeds as follows.

- $\overline{\mathcal{V}}$, new set of indices, is chosen as a *Small Ruling Subset* of $\mathcal{B}$ in graph $\mathcal{G}$.
- For each cluster $s \in \overline{\mathcal{V}}$, select *any* node $r_s \in \texttt{Sources}_s$ as a *representative* of s.
- $\overline{\mathcal{COVER}}_{\overline{\mathcal{V}}} = \{\overline{\texttt{Cover}}_p | p \in \overline{\mathcal{V}}\}$, new cover, is chosen as a BFS forest w.r.t. set of nodes $\{r_s | s \in \overline{\mathcal{V}}\}$ in the sub-graph induced by nodes spanned by $\mathcal{MBFS}_{\mathcal{V}}$.
- $\overline{\mathcal{SOURCES}}_{\overline{\mathcal{V}}} = \{\overline{\texttt{Sources}}_p | p \in \overline{\mathcal{V}}\}$, new collection of clusters, is chosen so that each new cluster contains all sources of blocked clusters that are in the same cover tree, namely

$$\overline{\texttt{Sources}}_p = \overline{\texttt{Cover}}_p \bigcap \left\{ \bigcup \{\texttt{Sources}_s | s \in \mathcal{B}\} \right\}$$

An efficient *deterministic* algorithm for computing SRS distributively has been given in [AGLP88].

9 EXTRACT algorithm

Definition 9.1 (Input/Output) *Input* of EXTRACT is a collection of forests $\mathcal{MBFS}_{\mathcal{A}} = \{\texttt{Mbfs}_s | s \in \mathcal{A}\}$ w.r.t. a source clusters $\mathcal{A}$. *Output* of EXTRACT is a forest BFS, such that, for each node v,

$$Distance_v(\texttt{BFS}) = \min_{s \in \mathcal{A}} Distance_v(\texttt{Mbfs}_s) \qquad (8)$$

In other words, given a bunch of (intersecting) forests $\mathcal{MBFS}_{\mathcal{A}}$, we want to select a *Parent* pointer at each node, as one of the "best" among *Parent* pointers in those forests. The "quality" is measured in terms of the length of the path to a root in the resulting forest. This is essentially the most "naive" reduction from the multiple sources to the single source, outlined in subsection 3.2. This is (trivially) achieved by selecting, for each node, a neighbor q, such that

$$Distance(\texttt{Mbfs}_q) = \min_{s \in \mathcal{A}} Distance(\texttt{Mbfs}_s) \qquad (9)$$

and then choosing q as its parent in the forest BFS.

10 Complexity

10.1 Definitions of complexity

Definition 10.1 (Amortized Complexities) We define *amortized complexities* of *any* protocol π that performs BFS for distance d_i (i.e. can be be applied on the i's level of recursion) as follows.

c_i^{π}: is the *amortized communication*. This is the *maximum* number of messages sent by the protocol over a network edge.

t_i^{π}: is the *amortized time*. This is the $\frac{1}{d_i}$ fraction of the time complexity of the protocol.

g_i^π: This is the number of time that the protocol needs to synchronize.

If DIJKSTRA algorithm were applied on i's level, then $c_i = t_i = g_i = d_i$, since the algorithm sends d_i messages per edge, runs d_i^2 time, and needs to synchronize d_i times. The total complexities of BFS algorithm can be bounded as $O(E \cdot c_k)$ messages and $O(D \cdot t_k)$ time. If we were simply running DIJKSTRA without any recursion, we would have had $c_k = t_k = D$ and thus would have obtained $O(E \cdot D)$ bound on number of messages and $O(D^2)$ bound on time.

In this extended abstract, we will only show the final recurrence for the complexities of the MAIN-BFS algorithm on different levels of recursion. They will be denoted, respectively, as c_i^{main}, t_i^{strip}, g_i^{multi}. Now, denote $\gamma_i = c_i^{main} + t_i^{main} + g_i^{main}$. Define α such that $\alpha = O(z \cdot X \cdot 5^z \cdot \log V)$. (Here $z = \log_X V$.)

Claim 10.2 The amortized complexities γ_i satisfy

$$\gamma_{i+1} \leq \begin{cases} \log V & \text{if } i = 0 \\ \alpha \cdot (\gamma_i + \frac{d_{i+1}}{d_i}) & \text{if } i > 0 \end{cases} \qquad (10)$$

Let us choose X to satisfy $5^z = X$, in which case $X = V^{\frac{1}{\sqrt{\log_5 V}}}$. Solving the recurrence 10.2 yields

$$\gamma_k \leq V^{\frac{O(1)}{\sqrt[4]{\log_2 V}}} \qquad (11)$$

Thus, the total communication and time complexities of the algorithm denoted by C, T are bounded by

$$C \leq \gamma_k \cdot E = O(EV^{\frac{O(1)}{\sqrt[4]{\log V}}}) \qquad (12)$$

$$T \leq \gamma_k \cdot D = O(D^{1+\frac{O(1)}{\sqrt[4]{\log V}}}) \qquad (13)$$

Acknowledgments

The author is very grateful to David Peleg for his crucial contributions to this work. David has encouraged the author to work on the problem, and contributed major ideas to the MERGE algorithm in this paper. Thanks are also due to David Shmoys for pointing author's attention to Gabow's work on scaling.

References

[AG85] Baruch Awerbuch and Robert G. Gallager. Distributed bfs algorithms. In 26^{th} *Annual Symposium on Foundations of Computer Science*, IEEE, October 1985.

[AG87] Baruch Awerbuch and Robert G. Gallager. A new distributed algorithm to find breadth first search trees. *IEEE Trans. Info. Theory*, IT-33(3):315–322, May 1987.

[AGLP88] Baruch Awerbuch, Andrew Goldberg, Michael Luby, and Serge Plotkin. Fast deterministic distributed maximal independent set algorithm. December 1988. Unpublished manuscript.

[Awe85] Baruch Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, October 1985.

[Awe87] Baruch Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election and related problems. In *Proceedings of the* 19^{th} *Annual ACM Symposium on Theory of Computing*, pages 230–240, ACM, May 1987.

[DS80] Edsger W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Info. Proc. Lett.*, 11(1):1–4, August 1980.

[Fre85] Greg N. Frederickson. A single source shortest path algorithm for a planar distributed network. In *Proceedings of 2nd Symp. on Theoretical Aspects of Computer Science*, January 1985.

[Gab85] Harold N. Gabow. Scaling algorithms for network problems. *J. Comp. and Syst. Sci.*, 31(2):148–168, October 1985.

[Gal82] Robert G. Gallager. *Distributed Minimum Hop Algorithms.* Technical Report LIDS-P-1175, MIT Lab. for Information and Decision Systems, January 1982.

[Jaf80] Jeffrey Jaffe. Using signalling messages instead of clocks. 1980. Unpublished manuscript.

[Lub86] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15(4):1036–1053, November 1986.

[Lub88] Michael Luby. Removing randomness in parallel computation without a processor penalty. In 29^{th} *Annual Symposium on Foundations of Computer Science*, Comp. Soc. of the IEEE, IEEE, 1988.

[MRR80] John M. McQuillan, Ira Richer, and Eric C. Rosen. The new routing algorithm for the ARPANET. *IEEE Trans. Comm.*, 28(5):711–719, May 1980.

[Pel87] David Peleg. Fast leader elections algorithms. 1987. unpublished manuscript.

[PU88] David Peleg and Eli Upfal. A tradeoff between size and efficiency for routing tables. In *Proceedings of the* 20^{th} *Annual ACM Symposium on Theory of Computing*, pages 43–52, ACM, May 1988.