Theory of Computing Systems manuscript No.
(will be inserted by the editor)

A Note on the Parallel Runtime of Self-Stabilizing Graph Linearization

Dominik Gall · Riko Jacob · Andrea Richa · Christian Scheideler · Stefan Schmid · Hanjo Täubig

Received: date / Accepted: date

Abstract Topological self-stabilization is an important concept to build robust open distributed systems (such as peer-to-peer systems) where nodes can organize themselves into meaningful network topologies. The goal is to devise distributed algorithms where nodes forward, insert, and delete links to neighboring nodes, and that converge quickly to such a desirable topology, independently of the initial network configuration. This article proposes a new model to study the parallel convergence time. Our model sheds light on the achievable parallelism by avoiding bottlenecks of existing models that can yield

A preliminary version of this work without all proofs and simulations has been presented at the 9th Latin American Theoretical Informatics Symposium (LATIN) 2010.
This work is partly supported by the National Science Foundation (NSF grant CCF-0830704) and by the German Research Foundation (DFG project SCHE 1592/1-1 and SFB 901: On-the-Fly Computing)

D. Gall
Institut für Informatik, Technische Universität München, Boltzmannstr. 3, 85748 Garching, Germany E-mail: dominik.gall@mytum.de

R. Jacob
Institut für Informatik, Technische Universität München, Boltzmannstr. 3, 85748 Garching, Germany *Present address:* Eidgenössische Technische Hochschule (ETH) Zürich, Universitätstr. 6, 8092 Zürich, Switzerland E-mail: rjacob@inf.ethz.ch

A. Richa
Department of Computer Science and Engineering, Arizona State University, Box 878809, Tempe, AZ 85287-8809, USA E-mail: aricha@asu.edu

C. Scheideler
Institut für Informatik, Universität Paderborn, Fürstenallee 11, 33102 Paderborn, Germany E-mail: scheideler@upb.de

S. Schmid
Telekom Innovation Labs & Technische Universität Berlin, Ernst-Reuter-Platz 7, 10587 Berlin, Germany E-mail: stefan@net.t-labs.tu-berlin.de

H. Täubig
Institut für Informatik, Technische Universität München, Boltzmannstr. 3, 85748 Garching, Germany E-mail: taeubig@in.tum.de

a distorted picture. As a case study, we consider local graph linearization—i.e., how to build a sorted list of the nodes of a connected graph in a distributed and self-stabilizing manner. In order to study the main structure and properties of our model, we propose two variants of a most simple local linearization algorithm. For each of these variants, we present analyses of the worst-case and best-case parallel time complexities, as well as the performance under a greedy selection of the actions to be executed. It turns out that the analysis is non-trivial despite the simple setting, and to complement our formal insights we report on our experiments which indicate that the runtimes may be better in the average case.

Keywords Distributed algorithms · Distributed systems · Peer-to-peer systems · Self-stabilization · Overlay networks · Performance

1 Introduction

Open distributed systems such as peer-to-peer systems are often highly transient in the sense that nodes join and leave at a fast pace. In addition to this natural churn, parts of the network can be under attack, causing nodes to leave involuntarily. Thus, *robustness* is a prime concern in the design of such a system. Over the last years, researchers have proposed many interesting approaches to build robust overlay networks. A particularly powerful concept in this context is (distributed) *topological self-stabilization.* A topological self-stabilizing mechanism guarantees that:

1. By local neighborhood changes (i.e., by creating, forwarding and deleting links with neighboring nodes), the nodes will *eventually* form an overlay topology with desirable properties from *any* initial (and in our case: *connected*) topology; this is known as the *convergence property.* (The assumption that the initial topology is connected is the fundamental minimal requirement to manage any topology in a distributed manner, as no communication is possible between disconnected components. In order to reestablish connectivity, we assume an external mechanism such as trackers or well-known bootstrap peers.)
2. The system will also stay in a correct configuration provided that no external topological changes occur; this property is called the *closure property.*

In this article, we address one of the first and foremost questions in distributed topological self-stabilization: *How to measure the parallel time complexity?* We consider a very strong adversary who presents our algorithm with an arbitrary connected network. We want to investigate how long it takes until the topology reaches a (to be specified) desirable configuration. While several solutions have been proposed in the literature over the last years, these known models are inappropriate to adequately model parallel efficiency: either they are overly pessimistic in the sense that they can force the algorithm to work serially, or they are too optimistic in the sense that contention or congestion issues are neglected.

Our model is aware of bottlenecks in the sense that nodes cannot perform too much work per time unit. Thus, we consider our new model as a further step to explore the right level of abstraction to measure parallel execution times.

As a case study, we investigate the problem of *graph linearization* where nodes—initially in a set of arbitrary connected graph components—are required to *sort* themselves with respect to their identifiers. As the most simple form of topological self-stabilization, linearization allows to study the main properties of our model. As we will see in our analysis, graph linearization under our model is already non-trivial and reveals an interesting structure.

This article focuses on two natural linearization algorithms, such that the influence of the modeling becomes clear. For our analysis, we will assume the existence of some hypothetical schedulers. In particular, we consider schedulers that, for each *round*, make one of the following selections for the *actions/rules* (or synonymously: *parallel (independent) steps*) to execute: one scheduler always makes a best possible, one a random, and one a "greedy" selection. Since the schedulers are only used for the *complexity analysis* of the protocols proposed, for ease of explanation, we treat the schedulers as global entities and we make no attempt to devise distributed, local mechanisms to implement them.[1]

1.1 Related Work

The first article to study self-stabilization in the context of distributed computing was [11] by E. W. Dijkstra. After Dijkstra's seminal work on the token ring, researchers have investigated self-stabilization in many other domains such as *clock synchronization* or *fault containment*. In 1991, Awerbuch and Varghese [4] proved that every local algorithm can be made self-stabilizing if all nodes keep a log of the state transitions until the current state. For a general overview of the field, the reader is referred to [7,13,18].

Our article focuses on *topological self-stabilization*. The construction and maintenance of a given network structure is of prime importance in many distributed systems, for example in peer-to-peer computing [12,14,24]. In the technical report of the distributed hash table Chord [25], stabilization protocols are described which allow the topology to recover from certain degenerate situations. Unfortunately, however, no algorithms are given to recover from *arbitrary* states. Similarly, also skip graphs [2] can be repaired from certain states, namely states which resulted from node faults and inconsistencies due to churn.

In order to gain insights into how to construct or self-stabilize more complex topologies such as hypercubic networks, in the last years, researchers started to analyze line and ring networks. The *Iterative Successor Pointer Rewiring*

[1] In fact, most likely no such local mechanism exists for implementing the worst-case and best-case schedulers, while we believe that local distributed implementations that closely approximate—within a constant factor of the parallel complexity—the randomized and greedy schedulers presented here would not be hard to devise.

Protocol [10] and the *Ring Network* [24] organize the nodes in a sorted ring. Unfortunately, both protocols have a large runtime. In [1], Angluin et al. present an efficient asynchronous algorithm which takes an initially weakly connected pointer graph and constructs a linked list with low contention. However, their algorithm is not self-stabilizing. In a follow-up paper [3], a self-stabilizing algorithm is given which assumes that nodes initially have out-degree 1.

The question of how to efficiently build a certain network structure is also related to *resource discovery* [17], *leader election* [8], and *parallel sorting* [16] problems. For instance, [17] analyzes how processes in a initial weakly connected knowledge graph can learn the identities of all other processes, [8] gives a deterministic algorithm for leader election in an initially connected knowledge graph, and [16] proposes a sorting algorithm for a parallel pointer machine that builds a binary tree.

The works closest to ours are by Onus et al. [23] and by Clouser et al. [9]. In [23], a local-control strategy called *linearization* is presented for converting an arbitrary connected graph into a sorted list. However, the strategy allows a node to communicate with an arbitrary number of its neighbors, which can be as high as $\Theta(n)$ for n nodes and is not scalable. Clouser et al. [9] formulated a variant of the linearization technique for arbitrary asynchronous systems in which edges are represented as Boolean shared variables. Any node may establish an undirected edge to one of its neighbors by setting the corresponding shared variable to true, and in each time unit, a node can manipulate at most one shared variable. If these manipulations never happen concurrently, it would be possible to emulate the shared variable concept in a message passing system in an efficient way. However, concurrent manipulations of shared variables can cause scalability problems because even if every node only modifies one shared variable at a time, the fact that the other endpoint of that shared variable has to get involved when emulating that action in a message passing system implies that a single node may get involved in up to $\Theta(n)$ many of these variables in a time unit.

Recently, Jacob et al. [20] generalized insights gained from graph linearization [15] to two dimensions, and presented a self-stabilizing $O(n^3)$-time construction for Delaunay graphs. Moreover, for a local-checkable variant of a skip graph, a polylogarithmic maintenance algorithm has been described in [22], and a self-stabilizing variant of a Chord graph appears in [21]. These works study a simpler model for the parallel runtime complexity that ignores congestion.

1.2 Our Contributions

The contributions of this article are two-fold. First, we present an alternative approach to modeling scalability of distributed, self-stabilizing algorithms that does not require synchronous executions like in [23] and also gets rid of the scalability problems in [9, 23] therefore allowing us to study the parallel time complexity of the proposed linearization approaches. Concretely, in our model, an *independent set* of nodes participates in the parallel execution of the different actions of the given round.

Second, we propose two variants of a simple, local linearization algorithm. For each of these variants, we present extensive formal analyses of their worst-case and best-case parallel time complexities, i.e., the number of (parallel) steps until the nodes converge to a desired fixpoint topology, and study their performance under a random and a greedy selection of the actions to be executed. We also validate the behavior of these algorithms by experiments which complement our formal findings, and indicate that the runtimes may in fact be better in practice. Finally, this article discusses a particular situation that illustrates how the new model compares to others proposed in the literature.

1.3 Organization

The remainder of this article is organized as follows. In Section 2, we describe our setting and the graph linearization problem, and introduce our model for the parallel time complexity. Section 3 presents a self-stabilizing algorithm together with a formal analysis. We report on our simulation results in Section 4. After discussing our approach and comparing our model to alternative frameworks in Section 5, we conclude the article in Section 6.

2 Model

We are given a system consisting of a fixed set V of n nodes. Every node has a unique (but otherwise arbitrary) and constant integer *identifier*. In the following, if we compare two nodes u and v using the notation $u < v$ or $u > v$, we mean that the identifier of u is smaller than v or vice versa. For any node v, $\text{pred}(v)$ denotes the predecessor of v (i.e., the node $u \in V$ of largest identifier with $u < v$) and $\text{succ}(v)$ denotes the successor of v according to "$<$". Two nodes u and v are called *consecutive* if and only if $u = \text{succ}(v)$ or $v = \text{succ}(u)$.

Connections between nodes are modeled as shared variables. Each pair (u, v) of nodes shares a Boolean variable $e(u, v)$ which specifies an undirected adjacency relation: u and v are called *neighbors* if and only if this shared variable is true.

The set of neighbor relations defines an *undirected graph* or *network* $G = (V, E)$ among the nodes. A variable $e(u, v) \in E$, a *link* between u and v (in the following, sometimes simply referred to as an undirected link $\{u, v\}$), can only be changed by u and v, and both u and v have to be involved in order to change $e(u, v)$. (More details on this will be given below.) For any node $u \in V$, let $u.L$ denote the set of left neighbors of u—the neighbors which have smaller identifiers than u—and $u.R$ the set of right neighbors of u.

In this article, $\deg(u) = |u.L \cup u.R|$ will denote the degree of a node u. Moreover, the distance between two nodes $\text{dist}(u, v)$ is defined as $\text{dist}(u, v) = |\{w : u < w \leq v\}|$ if $u < v$ and $\text{dist}(u, v) = |\{w : v < w \leq u\}|$ otherwise. The length of an edge $e = \{u, v\} \in E$ is defined as $\text{len}(e) = \text{dist}(u, v)$.

We consider *distributed algorithms* which are run by each node in the network. The algorithm or program executed by each node consists of a set of *variables* and *actions* (often also referred to as *rules*). An action has the form

$$< \texttt{name} > \quad : \quad < \texttt{guard} > \quad \rightarrow \quad < \texttt{commands} >$$

where $< \texttt{name} >$ is an *action label*, $< \texttt{guard} >$ is a Boolean predicate over the (local and shared) variables of the executing node and $< \texttt{commands} >$ is a sequence of commands that may involve any local or shared variables of the node itself or its neighbors. Given an action A, the set of all nodes involved in the commands is denoted by $V(A)$. Every node that either owns a local variable or is part of a shared variable $e(u, v)$ accessed by one of the commands in A is part of $V(A)$. Two actions A and B are said to be *independent* if $V(A) \cap V(B) = \emptyset$. For an action execution to be scalable we require that the number of operations involving interactions between the nodes (and therefore $|V(A)|$) is independent of n.

An action is called *enabled* if and only if its guard is true. Every enabled action is passed to some underlying scheduling layer (to be specified below). The scheduling layer decides whether to accept or reject an enabled action. If it is accepted, then the action is executed by the nodes involved in its commands.

We model distributed computation as follows. The assignments of all local and shared variables define a *system configuration.* Since our algorithms consider only variables that directly effect the topology, a configuration represents a *graph.* Hence, in the following, we will often treat the terms *graph*, *topology*, and *configuration*, as synonyms.

A *computation or execution* is a sequence of configurations, such that for each configuration c_i (a graph) at the beginning of *(computation) step* i, the next configuration c_{i+1} (the next graph) is obtained after executing an action that was selected by the scheduling layer in step i.

The concepts of sequences of configurations and of steps are useful to reason about the correctness of the self-stabilizing algorithm. In order to study the parallel time complexity, we additionally define the concept of a *(parallel time) round*: In each round, the scheduling layer may select any set of independent, enabled actions to be executed by the nodes, that is, a round consists of a set of parallel steps. Indeed, for the runtime analysis, we may think of the independent steps executed in parallel in a round as simultaneous.

Finally, the *work* performed (e.g., per round) is defined to the number of actions selected by the scheduling layer (in that round).

The following definition summarizes these concepts.

Definition 1 (Step, Round, Work) An enabled rule which is chosen and executed by the scheduler is called a *(computation) step.* The set of independent rules selected and executed in parallel by the scheduler constitutes a *(parallel time) round* (a set of parallel steps). The *work* performed in a round is defined to the number of actions selected by the scheduling layer (e.g., in that round).

In this paper, we will typically think of the *execution* as a sequence of *configurations* (i.e., topologies) following each as *(computation) steps.* In other words, the configuration c_i is obtained from c_{i-1} by the execution of step s_{i-1}.

The following definition summarized the self-stabilization requirements (see also Chapter 2.2 in [13]).

Definition 2 (Self-stabilizing Algorithm) A *self-stabilizing* distributed algorithm can be started in any arbitrary configuration (topology) and will eventually exhibit a desired *legal* (or *safe*) behavior. We define the desired legal behavior as a set of legal executions LE (for a particular system and a task). Every system execution should have the suffix that appears in LE. A configuration c is safe with regard to a task LE and an algorithm if every fair execution of the algorithm that starts from c belongs to LE; an algorithm is *self-stabilizing* for a task LE if every fair execution of the algorithm reaches a safe configuration with relation to LE (*convergence*) and stays there (*closure*).

Notice that our model can cover arbitrary asynchronous systems in which the actions are implemented so that the sequential consistency model applies (i.e., the outcome of the executions of the actions is equivalent to a sequential execution of them) as well as parallel executions in synchronous systems. In a round, the set of enabled actions selected by the scheduler must be independent as otherwise a configuration transition from one round to another would, in general, not be unique, and further rules would be necessary to handle dependent actions that we want to abstract from in this article.

2.1 Linearization

In this article, we are interested in designing distributed algorithms that can transform any connected component of an initial graph $G_0 = (V, E_0)$ into a sorted list (according to the node identifiers) using only local interactions between the nodes.

A distributed algorithm is called *(topologically) self-stabilizing*, if for any initial configuration or topology $G_0 = (V, E_0)$, it eventually arrives at a configuration $G_L = (V, E_L)$ in which the nodes are each connected component form a sorted list. In the following, for ease of presentation, we will typically assume that G_0 forms a *single* connected component: if a graph consists of multiple connected components, G_0 can be a placeholder for any of these connected components. The components are treated completely independently by our algorithms.

Definition 3 (Linear/Chain Graph G_L) Given a set of nodes V, the *linear/chain graph* G_L is defined as $G_L = (V, E_L)$ such that $\{u, v\} \in E_L$ if and only if

$$e(u, v) = 1 \quad \Leftrightarrow \quad u = \text{succ}(v) \ \vee \ v = \text{succ}(u)$$

Once the self-stabilizing algorithm arrives at this configuration (the legal configuration), it should stay there, i.e., the configuration is a *fixpoint* of the

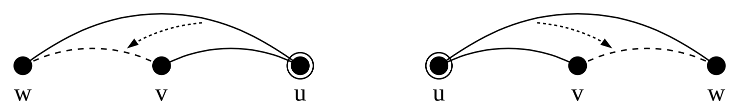


Fig. 1 Left and right linearization: node u forwards the link to its neighbor without violating connectivity.

algorithm. In the distributed algorithms studied in this article, each node $u \in V$ repeatedly performs simple *linearization steps* in order to arrive at that fixpoint.

Note that for a given connected initial graph $G_0 = (V, E_0)$, its linearized graph is unique.

Lemma 1 *$G_L = (V, E_L)$ is uniquely defined for a given node set V.*

Proof Definition 3 requires that $e(u, v) = 1$ if and only if $u = \text{succ}(v) \vee v = \text{succ}(u)$. Since nodes V have unique identifiers, for any given node $v \in V$, its successor is uniquely defined. □

Definition 4 (Linearization) A *linearization algorithm* is a distributed self-stabilizing algorithm (according to Definition 2) where

1. an *initial configuration* $c_1 \subseteq C$ forms *any (undirected) connected graph* $G_0 = (V, E_0)$,
2. the only *legal configuration* $L = \{c_l\} \subseteq C$ is the linear topology $G_L = (V, E_L)$ on the nodes V (i.e., E_L connects consecutive nodes, see Definition 3), and
3. actions only update the neighborhoods of the nodes (in our case, left and right linearization steps).

Our linearization algorithms will be based on simple linearization rules. A linearization involves three nodes u, v, and w with the property that u is directly connected to v and w and either $u < v < w$ or $w < v < u$. In both cases, u may command the nodes to move the edge $\{u, w\}$ to $\{v, w\}$. If $u < v < w$, this is called a *right* linearization and otherwise a *left* linearization step (see also Figure 1). Since only three nodes are involved in such a linearization step, this can be formulated by a scalable action. In the following, we will also call u, v, and w a *linearization triple* or simply a *triple.*

2.2 Schedulers

Our goal is to find linearization algorithms that spend as little time and work as possible in order to arrive at a sorted list. In order to investigate their worst, average, and best performance under concurrent executions of actions, we consider different schedulers. Essentially, the scheduler chooses a set of enabled actions from the fire-table and executes the corresponding steps in parallel, thus defining on how configuration (or topology) c_i becomes configuration c_{i+1}.

1. **Worst-case scheduler $\mathcal{S}_{\mathbf{wc}}$:** This scheduler must select a maximal independent set of enabled actions in each round, but it may do so to enforce a runtime (or work) that is as large as possible.
2. **Randomized scheduler $\mathcal{S}_{\mathbf{rand}}$:** This scheduler considers the set of enabled actions in a random order and selects, in one round, every action that is independent of the previously selected actions in that order.
3. **Greedy scheduler $\mathcal{S}_{\mathbf{greedy}}$:** This scheduler orders the nodes according to their degrees, from maximum to minimum. For each node that still has enabled actions left that are independent of previously selected actions, the scheduler picks one of them in a way specified in more detail later in this article when our self-stabilizing algorithm has been introduced. (Note, that 'greedy' refers to a greedy behavior w.r.t. the degree of the nodes; large degrees are preferred. Another meaningful 'greedy' scheduler could favor triples with largest gain w.r.t. the potential function that sums up all link lengths.)
4. **Best-case scheduler $\mathcal{S}_{\mathbf{opt}}$:** The enabled actions are selected in order to minimize the runtime (or work) of the algorithm. (Note that 'best' in this case requires maximal independent sets although there might be a better solution without this restriction.)

The worst-case and best-case schedulers are of theoretical interest and allow us to explore the parallel time complexity of the linearization approach. The greedy scheduler is a concrete algorithmic selection rule that we mainly use in the analysis as a lower bound on the performance under a best-case scheduler.

The randomized scheduler allows us to investigate the average case performance when a local-control randomized symmetry breaking approach is pursued in order to ensure sequential consistency while selecting and executing enabled actions.

As noted in the introduction, for ease of explanation, we treat the schedulers as global entities and we make no attempt to formally devise distributed, local mechanisms to implement them (that would in fact be an interesting, orthogonal line for future work). The schedulers are used simply to explore the parallel time complexity limitations (e.g., worst-case, average-case, best-case behavior) of the linearization algorithms proposed. In practice the algorithms LIN_{all} and LIN_{max} to be presented below may rely on any local-control rule (scheduler) to decide on a set of locally independent actions—which trivially leads to global independence—to perform at any given time.

3 Algorithms and Analysis

We now introduce our distributed and self-stabilizing linearization algorithms LIN_{all} and LIN_{max}. Section 3.1 specifies our algorithms formally and gives correctness proofs. Subsequently, we study the algorithms' runtime.

3.1 LIN$_{\text{all}}$ and LIN$_{\text{max}}$

Both algorithms LIN$_{\text{all}}$ and LIN$_{\text{max}}$ are based on two simple linearization rules: *linearize right* and *linearize left*. (Figure 1 visualizes the corresponding `commands` of these rules.) However, the two algorithms differ in the *preconditions* (i.e., the `guards`) when the rules are enabled.

In LIN$_{\text{all}}$, each node $v \in V$ enables the linearization rules for ***all*** possible triples that are incident to v. More formally, for every node u, we have the following rules *for every pair of neighbors v and w*:
linearize left(v,w): $(v, w \in u.L \ \wedge \ w < v < u) \ \rightarrow \ e(u,w) := 0, \ e(v,w) := 1$
linearize right(v,w): $(v, w \in u.R \wedge u < v < w) \ \rightarrow \ e(u,w) := 0, \ e(v,w) := 1$

LIN$_{\text{max}}$ is similar to LIN$_{\text{all}}$, but instead of proposing all possible triples on each side to the scheduler, LIN$_{\text{max}}$ only proposes the triple which is the furthest on the corresponding side.

In LIN$_{\text{max}}$, every node $u \in V$ uses the following rules *for every pair of neighbors v and w*:
linearize left(v,w): $(v, w \in u.L) \wedge (w < v < u) \wedge (\nexists x \in u.L \setminus \{w\} : x < v)$
$\rightarrow \ e(u,w) := 0, \ e(v,w) := 1$
linearize right(v,w): $(v, w \in u.R) \wedge (u < v < w) \wedge (\nexists x \in u.R \setminus \{w\} : x > v)$
$\rightarrow \ e(u,w) := 0, \ e(v,w) := 1$

We first show a basic property of LIN$_{\text{all}}$ and LIN$_{\text{max}}$.

Lemma 2 *Let $G_0 = (V, E_0)$ be an initial configuration (or graph), and let $G_t = (V, E_t)$ be the graph computed by LIN$_{all}$ or LIN$_{max}$, in step t. Then, it holds that if G_0 is connected, also G_t is connected.*

Proof We will prove the lemma by induction over the execution, i.e., over the sequence of configurations and steps. Concretely, for the induction, we will prove that if the configuration G_t describes a connected topology, then after step t the configuration G_{t+1} will again be connected.

Also note that while the rules of LIN$_{\text{all}}$ and LIN$_{\text{max}}$ differ in their preconditions (i.e., *guards*), their commands are the same and consist only of *left and right linearization steps*. Thus, in order to study the system's configuration transitions, we can focus on the left and right linearization steps, and do not have to differentiate between LIN$_{\text{all}}$ and LIN$_{\text{max}}$.

Initially, before step $t = t_0$, the network is connected by our assumption (*connected configuration*). We show that a linearization step of LIN$_{\text{all}}$ and LIN$_{\text{max}}$ at any time $t > t_0$ will not disconnect the graph. Without loss of generality, consider a triple $u, v, w \in V$ with $u < v < w$, $\{u, v\} \in E$, and $\{u, w\} \in E$ (cf Figure 1), which is right-linearized. (The proof for left-linearizations follows from symmetry arguments.) Clearly, the addition of a new edge cannot disconnect the network, and hence, it suffices to study the effect of removing the edge $e := \{u, w\}$ from E. Consider two arbitrary distinct nodes $x, y \in V$ that were connected before the linearization step. If there is a path between x and y that does not use e, then this path also exists after the linearization step, and connectivity is preserved. On the other hand, if all paths between x and y use e, then x and y must still be connected as well, as e can be emulated by the edges $\{u, v\}$ and $\{v, w\}$. Thus, the resulting configuration (i.e., topology) is connected again, and the claim follows. □

We can now prove that LIN_{all} and LIN_{max} are correct in the sense that eventually, a linearized graph will be reached.

Theorem 1 *LIN_{all} and LIN_{max} are self-stabilizing linearization algorithms.*

Proof According to Definition 2, a self-stabilizing algorithm must guarantee that starting from any configuration, (1) the system will *eventually* reach a correct configuration (*convergence*), and (2) the system will also stay in a correct configuration provided that no fault occurs (*closure*). In particular, a linearization algorithm (Definition 4) converts *any initially connected network* into a sorted chain.

Closure: We know from Lemma 1 that linearization specifies a *single* legal configuration: the linear graph G_L where consecutive nodes are connected (Definition 4). To show the closure property, we must prove that the linear graph constitutes a *fixpoint* of LIN_{all} and LIN_{max}: in the legal configuration, all actions are disabled and hence neighborhoods remain unchanged. Recall that there only exist two types of rules for LIN_{all} and LIN_{max}: *linearize left* and *linearize right*. For the sake of contradiction, assume that at least one such rule is still enabled in the unique legal configuration. Let us examine the two actions in turn: If a *linearize left* rule is still enabled, there must exist a node u having two neighbors v and w with $v, w \in u.L$ where $w < v < u$. However, this is a contradiction to the property that in a legal configuration, node u can have at most one predecessor ($|u.L| \leq 1$). Similarly, for a *linearize right* rule, a node u needs two neighbors v and w with $v, w \in u.R$ and $u < v < w$. This contradicts the assumption of a linearized topology where u can have at most one successor ($|u.R| \leq 1$). All actions must hence be disabled, and the linearized topology will remain unchanged.

Convergence: Let us now examine the convergence property. First note that if the network is in a configuration where it is connected but it does not constitute the linear chain graph yet, then there must exist a node having at least two left neighbors or at least two right neighbors. Accordingly, the *linearize left* or *linearize right* rule is enabled and will continue to change the topology in this step.

To show eventual convergence to the unique legal configuration (the linear graph), we will prove that after any execution of the *linearize left* or *linearize right* rule, the topology will come "closer" to the linearized configuration, in the following sense: we can define a potential function whose value is monotonically decreased with each executed rule (and hence an arbitrary sequence of interleaved executions), and which is minimized for the linearized network topology.

We consider the potential function Ψ that sums up the lengths (hop distances) of all existing links with respect to the linear ordering of the nodes, i.e., $\Psi = \sum_{e \in E} \text{len}(e)$. Due to our assumptions, in *any* initial configuration, it holds that the network G_0 is connected, and hence $\Psi \geq n-1$: a connected graph consists of at least $n-1$ edges.

Whenever an action is executed (in our case: a linearization step is performed), the potential Ψ is reduced by at least the length of the shorter edge in the linearization triple, i.e., by $\text{len}(\{u,v\}) \geq 1$ (see Figure 1): edge $\{u,w\}$ is removed and potentially an edge $\{v,w\}$ inserted (if it does not exist already). Thus, LIN_{all} and LIN_{max} will eventually reach a topology of minimal potential Ψ, where all actions are disabled. We know from Lemma 2 that LIN_{all} or LIN_{max} will never disconnect an initially connected graph again. However, the only connected topology with minimum Ψ (i.e., minimum edge lengths) is the desired legal configuration (the line topology).

Thus, there is always an action enabled unless the graph reached the target topology. Therefore, the network converges to a line in a finite number of steps, and the claim follows. □

3.2 Runtime

We first study the worst case scheduler $\mathcal{S}_{\text{wc}}$ for both LIN_{all} and LIN_{max}.

Theorem 2 *Under a worst-case scheduler $\mathcal{S}_{wc}$, LIN_{max} terminates after $O(n^2)$ work (single linearization steps), where n is the total number of nodes in the system. This is tight in the sense that there are situations where under a worst-case scheduler $\mathcal{S}_{wc}$, LIN_{max} requires $\Omega(n^2)$ rounds.*

Proof **Upper Bound:** In order to study the evolution of the configurations (i.e., topologies) over time (i.e., over the execution), we define a potential function over the topology. Let $\zeta_l(v)$ denote the length of the longest edge out of node $v \in V$ to the left and let $\zeta_r(v)$ denote the length of the longest edge out of node v to the right. If node v does not have any edge to the left, we set $\zeta_l(v) = \frac{1}{2}$, and similarly for the right. We consider the potential function Φ which is defined as

$$\Phi = \sum_{v \in V} \zeta_l(v) + \zeta_r(v).$$

Let Φ_i denote the value of Φ after i time steps. Observe that initially, $\Phi_0 \leq n^2$, as $\zeta_l(v) + \zeta_r(v) \leq n$ for each node v. We show that after i linearization steps, the potential is at most $\Phi_i \leq n^2 - \frac{i}{2}$. Since LIN_{max} terminates (cf also Theorem 1) with a potential $\Phi_j > 0$ for some j (the term of each node is positive), the claim follows. In order to see why the potential is reduced by at least $\frac{1}{2}$ in every step, consider a triple u, v, w which is right-linearized and where $u < v < w$, $\{u,v\} \in E$, and $\{u,w\} \in E$. (Left-linearizations are similar and not discussed further here.) During the linearization step, $\{u,w\}$ is removed from E and the edge $\{v, w\}$ is added if it did not already exist. We are interested in the change of the value, of Φ, i.e., $\Delta\Phi = \Delta\zeta_r(u)+\Delta\zeta_r(v)+\Delta\zeta_l(w)$. Since the rightmost neighbor of u changes from w to v, we know $\Delta\zeta_r(u) = \text{dist}(u, v) - \text{dist}(u, w) = -1$. Further, we know that $\zeta_r(v)$ changes its value only if there was no edge $\{v, w\}$ before the linearization step. Thus, we have $\Delta\zeta_r(v) \leq \text{dist}(v, w) - \frac{1}{2} = 1 - \frac{1}{2} = \frac{1}{2}$. Since w had a left neighbor (u) before the linearization, the value of $\zeta_l(w)$ cannot increase, i.e., $\Delta\zeta_l(w) \leq 0$. This implies $\Delta\Phi = \Delta\zeta_r(u) + \Delta\zeta_r(v) + \Delta\zeta_l(w) \leq -1 + \frac{1}{2} + 0 = -\frac{1}{2}$. Since at least one triple can be linearized in every round, this concludes the proof.

Lower Bound: We consider a simple network over a set of nodes $V = \{1, \ldots, n\}$, and show that there is a scheduling strategy for this network that creates a large number of blocked nodes in each round, ending up with only constant work per round and a quadratic number of rounds. Our sample network resembles a complete bipartite graph where the first half of all nodes is completely connected to the second half (see Figure 2). In addition, all nodes are adjacent to their predecessors and successors, i.e., all links of the desired linearized topology are already present. (During linearization, one link will disappear in each step.)

Now consider a node having an incident edge which is a longest link for some other node. Note that initially, only the leftmost and the rightmost node (if nodes are ordered with respect to their IDs) fulfill this property (the longest edges of the nodes on the right all end at the leftmost node, and vice versa). In the following, we will count the number of longest left and right links incident at a node $v \in V$ and will denote such a link a (left-link or right-link) *pebble*. For instance, in Figure 2, node 1 has the longest left-link pebbles of nodes $n/2 + 1, \ldots, n$, whereas node n has the longest right-link pebbles of nodes $1, \ldots, n/2$. In the first round, the scheduler decides to (left-)linearize node $n/2 + 1$ (which automatically involves nodes 1 and 2 according to LIN_{max}) and to right-linearize node $n/2$ (which automatically involves nodes $n - 1$ and n). Observe that these two actions block all other linearization steps since any other triple would involve some non-blocked node having a pebble, but nodes 1 and n are the only nodes with pebbles and are blocked. Therefore, in the first round, the edges $\{n/2, n\}$ and $\{1, n/2 + 1\}$ are removed, and the longest left-link pebble of node $n/2 + 1$ is moved from node 1 to node 2 and the longest right-link pebble of node $n/2$ is moved from node n to node $n - 1$.

In the next round, the scheduler decides to left-linearize node $n/2 + 2$ and to right-linearize node $n/2 - 1$. Again, this involves nodes 1 and 2, as well as nodes $n - 1$ and n. Therefore all nodes with pebbles are blocked which prevents any further action. Besides removing the respective edges the effect of the round is that the longest left-link pebble of node $n/2 + 2$ moves from node 1 to node 2 and the longest right-link pebble of node $n/2 - 1$ moves from node n to node $(n - 1)$. This procedure is repeated until all longest left-link pebbles (except the one of node n) have moved from node 1 to node 2 and all longest right-link pebbles (except the one of node 1) have moved from node n to node $n - 1$. The length of this first phase is $n/2$ rounds. Note that there are always exactly two linearization triples in each round (except for the last

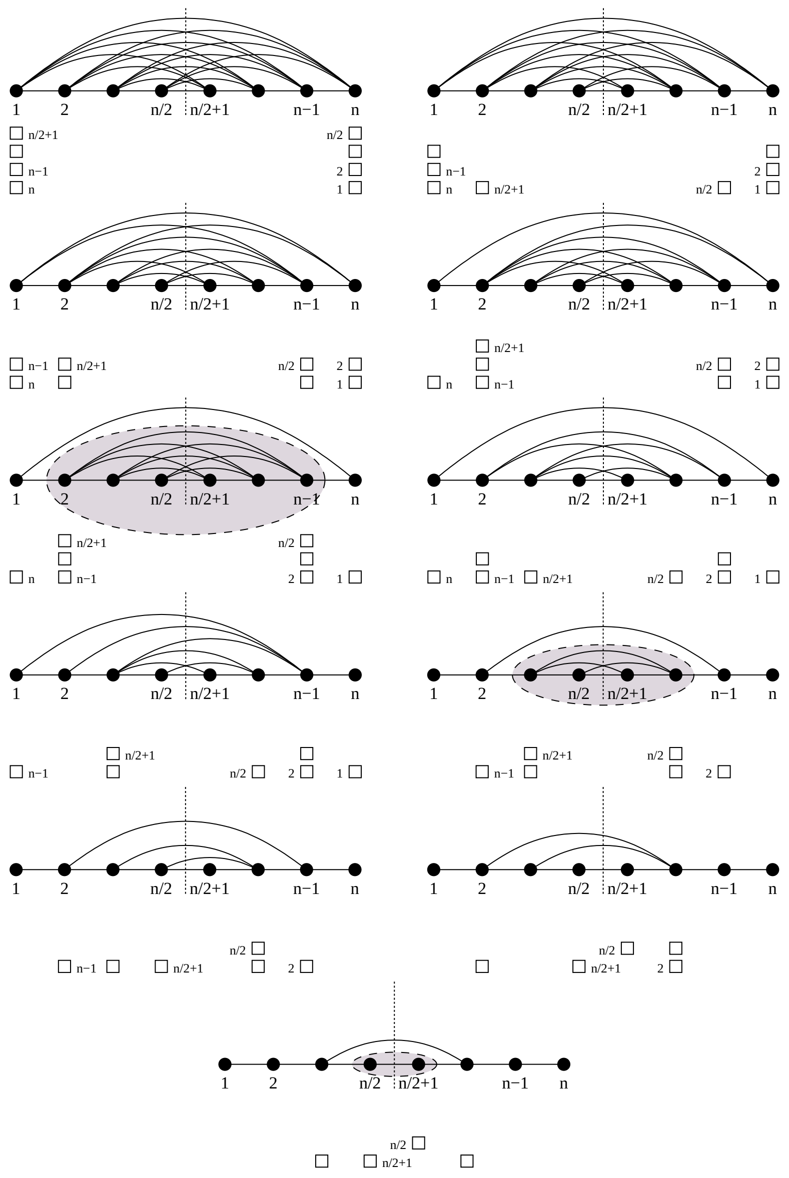


Fig. 2 Bad case for linearizing a complete bipartite network. See the proof of Theorem 2 for explanations.

two rounds, where only one triple is linearized). At the end of this first phase, there is one link left from node 1 to node n, which is later linearized in parallel to the next phase. At this point, the scheduler has created again a complete bipartite network, which is smaller by one node on both sides.

In applying the same method recursively, the scheduler implements a series of *phases* where in each phase all longest left-link pebbles (except one) move one node to the right and all longest right-link pebbles (except one) move one node to the left (one left-pebble and one right-pebble per round). At the end of the phase, only one triple of the inner part can be linearized. At this time, the single outer edge is also linearized (in all rounds before, both of the outmost nodes of the inner part are blocked, therefore this large edge persists until then). Such a Phase i takes $n/2+1-i$ rounds. The total number of rounds is thus at least

$$\sum_{i=1}^{n/2}\left(\frac{n}{2}+1-i\right)=\sum_{i=1}^{n/2} i \in \Omega(n^2).$$

□

For the LIN$_{\mathrm{all}}$ algorithm, we obtain a slightly higher upper bound, as we will show next. In the analysis, we need the following helper lemma.

Lemma 3 *Let Ψ be any positive potential function, where Ψ_0 is the initial potential value and Ψ_i is the potential after the i^{th} round of a given algorithm ALG. Assume that $\Psi_i \leq \Psi_{i-1}\cdot(1-1/f)$ and that ALG terminates if $\Psi_j \leq \Psi_{stop}$ for some $j \in \mathbb{N}$. Then, the runtime of ALG is at most $O(f \cdot \log(\Psi_0/\Psi_{stop}))$ rounds.*

Proof From $\Psi_i \leq \Psi_{i-1}\cdot(1-1/f)$, it follows that $\Psi_j \leq \Psi_0 \cdot (1-1/f)^j$.
Now consider $j = f \cdot \ln \frac{\Psi_0}{\Psi_{stop}}$, which leads to (using $\ln(1+x) \leq x$ for all $x > -1$)

$$\begin{aligned}\Psi_j &\leq \Psi_0 \cdot (1-1/f)^{f\cdot \ln \frac{\Psi_0}{\Psi_{stop}}} = \Psi_0 e^{f\cdot\left(\ln \frac{\Psi_{stop}}{\Psi_0}\right)\cdot \ln(1-1/f)} \\ &\leq \Psi_0 e^{f\cdot\left(\ln \frac{\Psi_0}{\Psi_{stop}}\right)\cdot(-1/f)} = \Psi_0 e^{-\ln \frac{\Psi_0}{\Psi_{stop}}} = \Psi_{stop}\end{aligned}$$

□

Theorem 3 *LIN$_{all}$ terminates after $O(n^2 \log n)$ many rounds under a worst-case scheduler $\mathcal{S}_{wc}$, where n is the network size.*

Proof We consider the potential function

$$\Psi = \sum_{e \in E} \mathrm{len}(e) \quad \text{with} \quad \Psi_0 \leq \binom{n}{2}(n-1) < n^3$$

We show that in each round, this potential is multiplied by a factor of at most $1-\Omega(1/n^2)$.

Consider an arbitrary triple $u, v, w \in V$ with $u < v < w$ which is right-linearized by node u. (The case of left-linearizations is similar and not discussed further here.) During a linearization step, the sum of the edge lengths is reduced by at least one. So what is the amount of blocked potential in a round due to the linearization of the triple (u, v, w) (cf also the proof of Theorem 4)? Nodes u, v, and w have at most $\deg(u)+\deg(v)+\deg(w) < n$ many independent neighbors, and hence, in the worst case, when the triple's incident edges are removed (the blocked potential is at most $O(n^2)$), these neighbors fall into different disconnected components which cannot be linearized further in this round; in other words, the remaining components form sorted lines. The blocked potential amounts to at most $\Theta(n^2)$. Thus, together with Lemma 3 (using $\Psi_0 < n^3$, $\Psi_{stop} \in O(n)$, $f = n^2$), the claim follows. □

Note that Theorem 3 suggests that allowing to linearize *any* neighbor (like LIN_{all}) in the independent sets may yield higher runtimes than restricting the selection to the maximal neighbor.

Besides $\mathcal{S}_{\text{wc}}$, we are interested in the following type of greedy scheduler. In each round, both for LIN_{all} and LIN_{max}, $\mathcal{S}_{\text{greedy}}$ orders the nodes with respect to their *remaining (total, i.e., left plus right) degrees*: after a triple has been fired, the three nodes' incident edges are removed. For each node $v \in V$ selected by the scheduler according to this order (which still has enabled actions left which are independent of previously selected actions), the scheduler greedily picks the enabled action of v which involves the two most distant neighbors on the side with the larger remaining degree. (If the number of remaining left neighbors equals the number of remaining neighbors on the right side, then an arbitrary side can be chosen.) The intuition behind $\mathcal{S}_{\text{greedy}}$ is that neighborhood sizes are reduced quickly in the linearization process.

Under this greedy scheduler, we get the following improved bound on the time complexity of LIN_{all}.

Theorem 4 *Under a greedy scheduler $\mathcal{S}_{greedy}$, LIN_{all} terminates in $O(n \log n)$ rounds, where n is the total number of nodes in the system.*

Proof Again, we consider the potential function $\Psi = \sum_{e \in E} \text{len}(e)$. As before, $\Psi_0 \leq n(n-1)^2/2$. At the end we have $\Psi_{stop} = n - 1$. We will prove that in each round, the potential is multiplied by a factor of at most $1 - 1/(24 \cdot n)$, i.e., $f(n) \leq 24n$. Given this factor bound and $\Psi_0/\Psi_{stop} < n^2$, Lemma 3 implies that the total number of rounds is in $O(n \log n)$.

It remains to prove that the potential is indeed reduced by a factor of $1 - \Theta(1/n)$ in each round. First, observe that firing a triple reduces the potential Ψ, but prevents other triples from being fired in the same round. For our analysis, we want to bound this blocked potential. Recall our definition of the greedy scheduler $\mathcal{S}_{\text{greedy}}$ which always chooses the node with the largest remaining degree and selects for the linearization operation the two neighbors which are furthest away from this node on the side of larger degree. Consider any triple $v_1, v_2, v_3 \in V$ of nodes with $v_1 < v_2 < v_3$ and $\{v_1, v_3\}, \{v_1, v_2\} \in E$ which is right-linearized (left-linearizations are similar and not described further here). As we will see, removing the edge $\{v_1, v_3\}$ and adding (if necessary) edge $\{v_2, v_3\}$ reduces Ψ by at least $\widehat{\deg}(v_1)/4$, where $\widehat{\deg}(v_1)$ is the number of neighbors of v_1 if the edges incident to the already processed nodes in this round by the greedy scheduler are removed: Note that by removing $\{v_1, v_3\}$ and possibly adding $\{v_2, v_3\}$, the potential is reduced by at least $\text{dist}(v_1, v_3) - \text{dist}(v_2, v_3) = \text{dist}(v_1, v_2)$. Since the potential decreases by at least 1, we know for the special cases of $\widehat{\deg}(v_1) = 2$ and $\widehat{\deg}(v_1) = 3$ that Ψ

decreases by more than $\widehat{\deg}(v_1)/4$. Furthermore, according to $\mathcal{S}_{\text{greedy}}$, v_1 has at least as many remaining neighbors on the right as it has on the left, i.e., we have that $\text{dist}(v_1, v_2) \geq \widehat{\deg}(v_1)/2 - 1$ (since v_2 and v_3 are the two most distant neighbors on the side having at least half of the neighbors). This implies for $\widehat{\deg}(v_1) \geq 4$ that Ψ decreases by at least $\widehat{\deg}(v_1)/2 - 1 \geq \widehat{\deg}(v_1)/4$.

By firing the triple, we may lose the option to linearize other nodes. In order to bound the blocked potential by this linearization step, we consider the components that remain after nodes v_1, v_2 and v_3 (plus incident edges) have been removed. Let w be an arbitrary neighbor of v_i, for $i \in \{1, 2, 3\}$. Consider the connected component after v_i has been removed which includes w. We distinguish two different cases.

Case 1: If this connected component forms a line where nodes are ordered, the nodes in the component cannot be linearized or scheduled further in this step. Thus, the component blocks the potential contained in this line, which is however at most n. Moreover, we lose the edge $\{v_i, w\}$ which also has a potential of at most n, yielding a total potential of at most $2n$.

Case 2: If the component has any other form, there must exist triples in it that can still be fired later in this round, and hence, the blocked potential is accounted for similarly during the linearization of another triple. Thus, we only have to take into account the blocked potential due to the lost edge incident to the triple which is at most n.

The total amount of blocked potential is therefore at most $6 \cdot \widehat{\deg}(v_1) \cdot n$: As $\mathcal{S}_{\text{greedy}}$ chooses the node with largest remaining degree, it holds that $\widehat{\deg}(v_1) \geq \max\{\widehat{\deg}(v_2), \widehat{\deg}(v_3)\}$. Since we have at most a blocked potential of $2n$ per neighbor of v_i, for $i \in \{1, 2, 3\}$, the blocked potential is at most $3 \cdot \widehat{\deg}(v_1) \cdot 2n$.

Since $\widehat{\deg}(v_1)/2 - 1 \geq \widehat{\deg}(v_1)/4$, we have that $\Psi_i \leq (1 - 1/(24 \cdot n))\Psi_{i-1} = (1 - \Theta(1/n))\Psi_{i-1}$, and the claim follows. □

Finally, we have also investigated an optimal scheduler $\mathcal{S}_{\text{opt}}$.

Theorem 5 *Even under an optimal scheduler $\mathcal{S}_{opt}$, both LIN$_{all}$ and LIN$_{max}$ require at least $\Omega(n)$ rounds in certain situations.*

Proof Let $v_1, v_2, \ldots, v_n \in V$ denote the nodes in sorted order, i.e., $v_1 < v_2 < \ldots < v_n$. Consider the following initial topology $G_0 = (V, E_0)$ where $\forall i$ such that $0 < i < n - 1$: $\{v_i, v_{i+1}\} \in E_0$. Additionally, E_0 contains a long edge $e := \{v_1, v_n\} \in E_0$. In the beginning, edge e has length of $\text{len}(e) = n - 1$. Observe that in each round, both for LIN$_{\text{all}}$ and LIN$_{\text{max}}$, the length of e is reduced by one. Thus, by induction, it takes at least a linear number of rounds to sort G_0, as the execution is inherently sequential. □

3.3 Degree Cap

It is desirable that the nodes' neighborhoods or degrees do not increase much during the sorting process. We investigate the performance of LIN$_{\text{all}}$ and

LIN_{max} under the following *degree cap model*. Observe that during a linearization step, only the degree of the node in the middle of the triple can increase (see Figure 1). We do not schedule triples if the middle node's degree would increase, with one exception: during left-linearizations, we allow a degree increase if the middle node has at most one left neighbor, and during right-linearizations we allow a degree increase to the right if the middle node has at most one right neighbor. In other words, we study a *degree cap of two*.

We find that both our algorithms LIN_{all} and LIN_{max} still terminate with a correct solution under this restrictive model.

Theorem 6 *With degree cap, LIN_{max} terminates in at most $O(n^2)$ many rounds under a worst-case scheduler $\mathcal{S}_{wc}$, where n is the total number of nodes in the system. Under the same conditions, LIN_{all} requires at most $O(n^3)$ rounds.*

Proof Bound for LIN_{max}: The claim follows from the same arguments as used in Theorem 2. It only remains to prove that in each round there exists a triple which can be right or left linearized. In order to see that at least one triple can be linearized, consider the node u of largest order which has two neighbors to the right. (If there does not exist any node with two neighbors that can be right-linearized, we apply the same argument to the left. If there is no node with two left neighbors that can be left-linearized, this implies that the graph is already sorted.)

Let v and w be u's two neighbors to the right, where $v < w$. The triple consisting of the three nodes u, v and w can definitely be right-linearized without violating the degree cap constraint: v is the only node whose degree increases during the linearization step. However, v's degree to the right cannot be more than two after linearization, otherwise we have a contradiction to our assumption that u is the largest node with two neighbors to the right.

Bound for LIN_{all}: We consider again the potential function $\Psi = \sum_{e \in E} \text{len}(e)$ summing up all edge lengths in the graph. Note that initially, $\Psi_0 < n^3$, and each linearization step reduces Ψ by at least one. When the graph is sorted, $\Psi < n$. Therefore, for the $O(n^3)$ bound, it remains to prove that the system cannot deadlock and there is progress in every round. However, this holds for the same reasons as discussed above for the LIN_{max} bound. □

Interestingly, as we will see in the experimental section (Section 4), the runtime of LIN_{all} and LIN_{max} is typically better than shown in Theorem 6. Moreover, it turns out that even without imposing a degree cap, LIN_{all} and LIN_{max} do not increase the maximal degrees during their computations.

4 Experiments

In order to improve our understanding of the parallel complexity and the behavior of our algorithms, we have implemented a simulation framework which allows us to study and compare different algorithms, topologies and schedulers. In this section, some of our findings will be described in more detail.

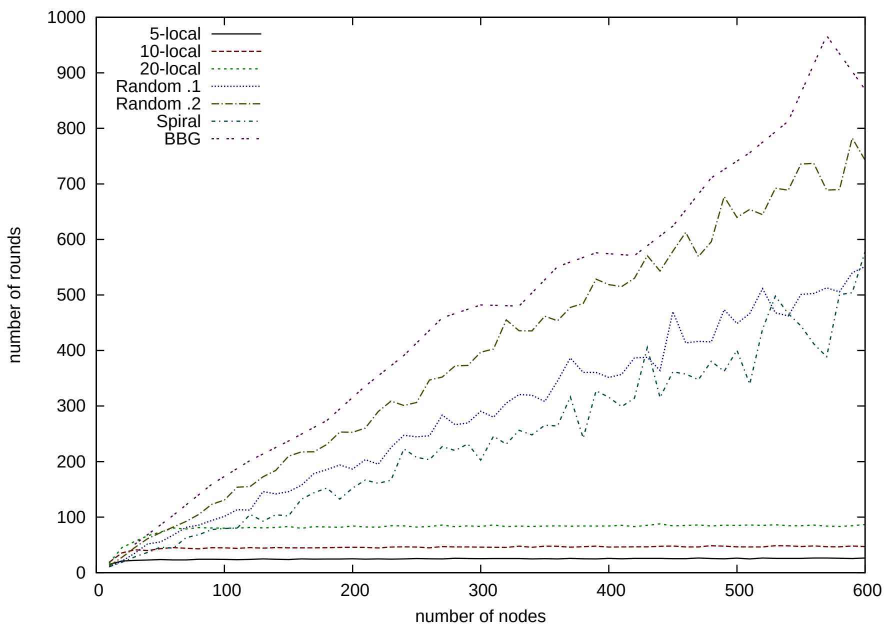
We will consider the following graphs. We chose these graphs as they appeared to be good representatives for easy, average, and difficult problem instances.

1. **Random graph:** Any pair of nodes is connected with probability p, i.e., if $V = \{v_1, \ldots, v_n\}$, then $\mathbb{P}[\{v_i, v_j\} \in E] = p$ for all $i, j \in \{1, \ldots, n\}$. If necessary, edges are added to ensure connectivity.
2. **Bipartite backbone graph (k-BBG):** For $n = 3k$ for some positive integer k define the following k-bipartite backbone graph on the node set $V = \{v_1, \ldots, v_n\}$. All n nodes are connected to their respective successors and predecessors (except for the first and the last node). This structure is called the graph's *backbone*. Additionally, there are all $(n/3)^2$ edges from nodes in $\{v_1, \ldots, v_k\}$ to nodes in $\{v_{2k+1}, \ldots, v_n\}$.
3. **Spiral graph:** The spiral graph $G = (V, E)$ is a sparse graph forming a spiral, i.e.,
   $V = \{v_1, \ldots, v_n\}$ where $v_1 < v_2 < \ldots < v_n$ and
   $E = \{\{v_1, v_n\}, \{v_n, v_2\}, \{v_2, v_{n-1}\}, \{v_{n-1}, v_3\}, \ldots, \{v_{\lceil n/2 \rceil}, v_{\lceil n/2 \rceil + 1}\}\}$.
4. **k-local graph:** This graph avoids long-range links. Let $V = \{v_1, \ldots, v_n\}$ where $v_i = i$ for $i \in \{1, \ldots, n\}$. Then, $\{v_i, v_j\} \in E$ if and only if $|i - j| \leq k$.

We will constrain ourselves to two schedulers here: the greedy scheduler $\mathcal{S}_{\text{greedy}}$ which we have already considered in the previous sections, and a randomized scheduler $\mathcal{S}_{\text{rand}}$ which among all possible enabled actions chooses one *uniformly at random* at a time, deletes all conflicting actions, and repeats until a maximal non-conflicting set of actions is chosen.

Many experiments have been conducted to shed light onto the parallel runtime of LIN_{all} and LIN_{max} in different networks. Figure 3 (top) depicts some of our results for LIN_{all}. As expected, in the k-local graphs, the execution is highly parallel and yields a "constant" runtime—independent of n. The sparse spiral graphs appear to entail an almost linear time complexity, and also the random graphs perform better than our analytical upper bounds suggest. Among the graphs we tested, the *BBG* network yielded the highest execution times. Figure 3 (bottom) gives the corresponding results for LIN_{max}.

A natural yardstick to measure the quality of a linearization algorithm—besides the parallel runtime—is the node degree. For instance, it is desirable that an initially sparse graph will remain sparse during the entire linearization process. It turns out that LIN_{all} and LIN_{max} indeed maintain a low degree. Figure 4 shows how the maximal and average degrees evolve over time both for LIN_{all} and LIN_{max} on two different random graphs. Note that the average degree cannot increase because the rules only move or remove edges. The random graphs studied in Figure 4 have a high initial degree, and it is interesting to analyze what happens in case of sparse initial graphs. Figure 5 plots the maximal node degree over time for the spiral graph. While there is an increase in the beginning, the degree is moderate at any time and declines again quickly.



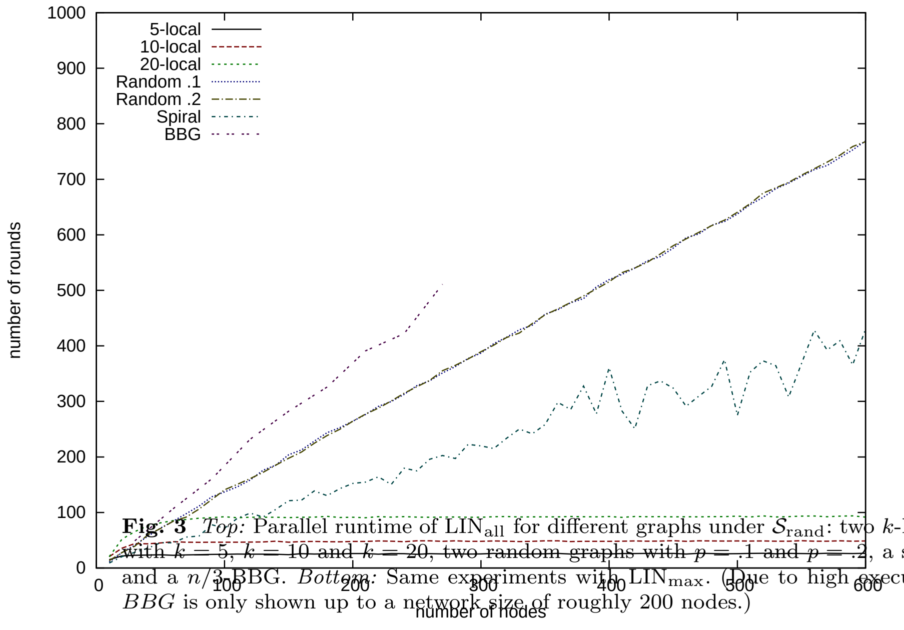
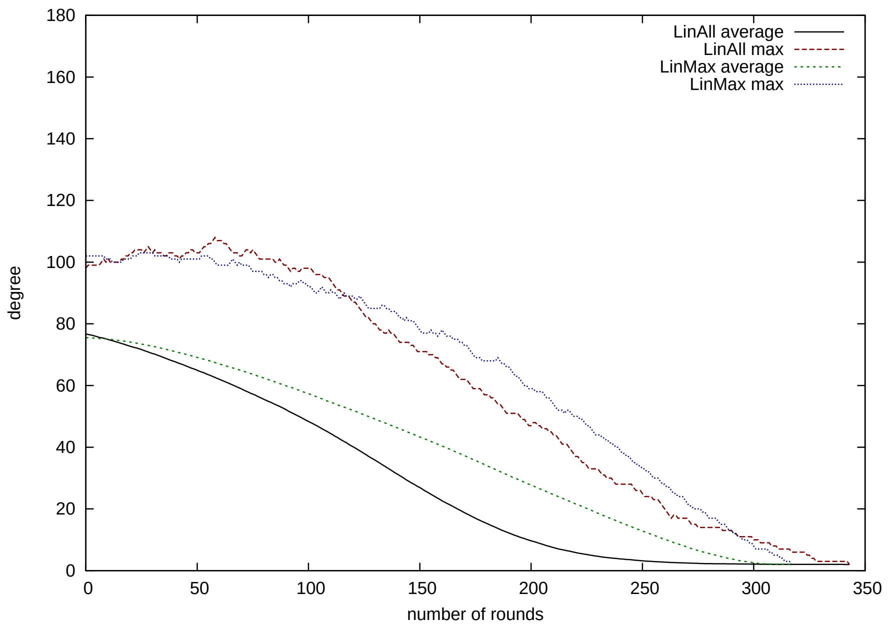


Fig. 3 *Top:* Parallel runtime of $\mathrm{LIN}_{\mathrm{all}}$ for different graphs under $\mathcal{S}_{\mathrm{rand}}$: two k-local graphs with $k = 5$, $k = 10$ and $k = 20$, two random graphs with $p = .1$ and $p = .2$, a spiral graph and a $n/3$ BBG. *Bottom:* Same experiments with $\mathrm{LIN}_{\mathrm{max}}$. (Due to high execution times, *BBG* is only shown up to a network size of roughly 200 nodes.)



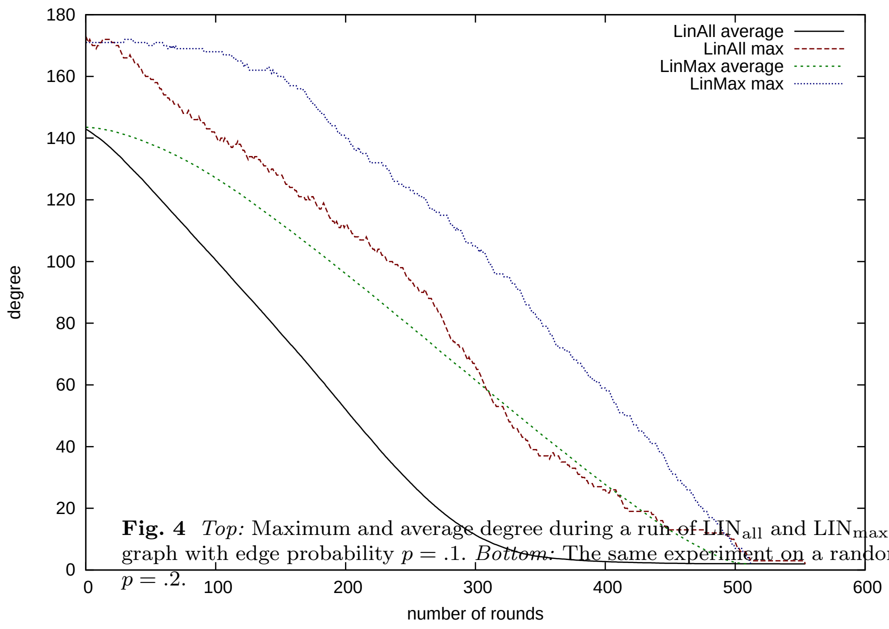


Fig. 4 *Top:* Maximum and average degree during a run of $\mathrm{LIN}_{\mathrm{all}}$ and $\mathrm{LIN}_{\mathrm{max}}$ on a random graph with edge probability $p = .1$. *Bottom:* The same experiment on a random graph with $p = .2$.

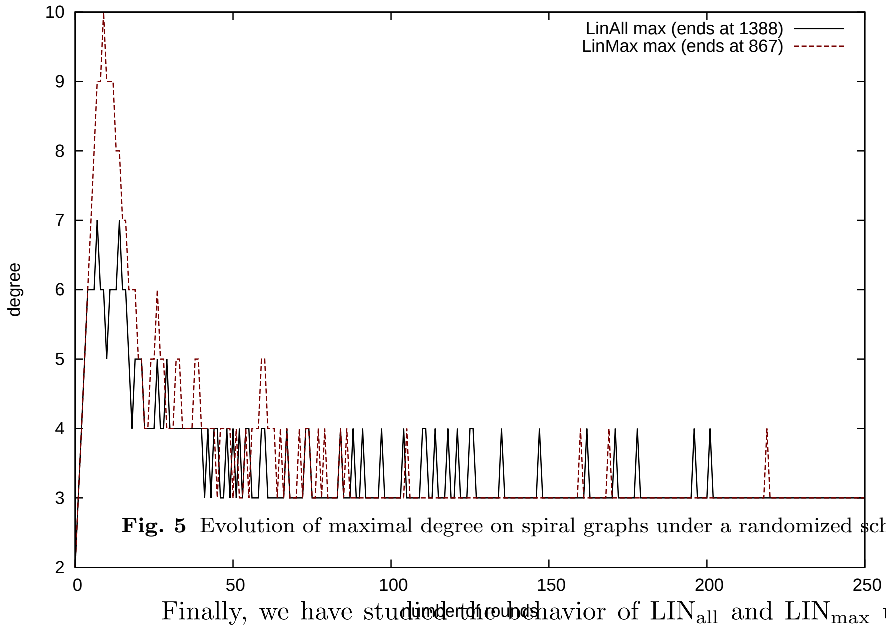


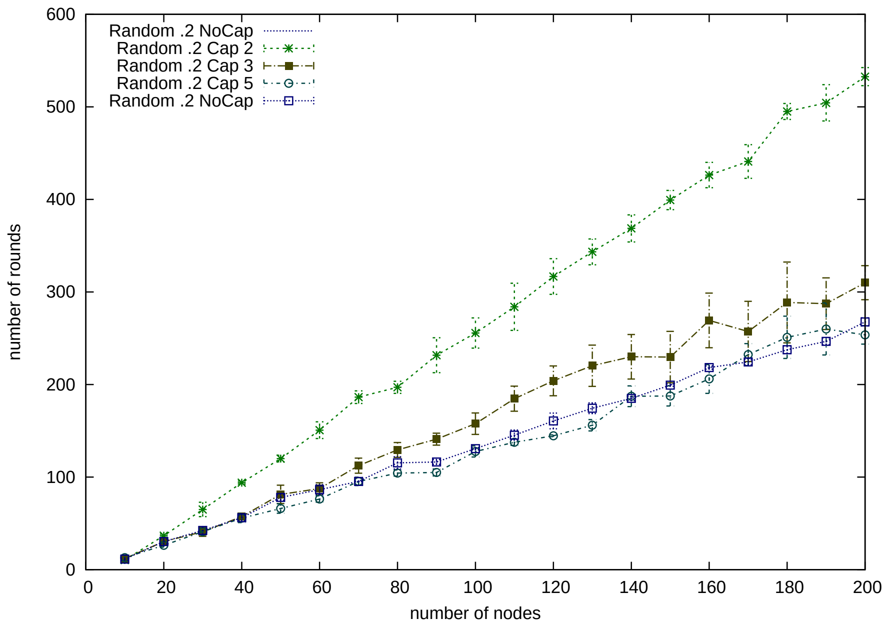
Fig. 5 Evolution of maximal degree on spiral graphs under a randomized scheduler $\mathcal{S}_{\text{rand}}$.

Finally, we have studied the behavior of LIN_{all} and LIN_{max} under a degree cap constraint, where triples can only be linearized if the center node's degree does not grow to more than a certain threshold in the corresponding direction. Figure 6 (top) indicates that all the runtime remains roughly linear even for a degree cap of two. For degree caps larger than two, the performance is better. However, interestingly, it seems that the number of rounds does not decrease monotonously with larger caps—rather, a lower degree cap might help to speed-up the linearization process under certain circumstances. Figure 6 (bottom) shows the runtimes under a *BBG* graph; here, the greedy scheduler requires much more (and even super-linear) time compared to the other settings, which indicates that this configuration together with the *BBG* graph is a particularly challenging one.

5 Discussion and Model Comparison

This section provides a short discussion and also compares our approach to the alternative models, e.g., to the so-called *critical path model* introduced in [5, 6].

One may wonder whether our actions (left and right linearization) really have to be executed in an independent way in order to maintain sequential consistency. Here, it would be sufficient if each node initiates a new linearization only after its previously initiated linearization has been completed (or



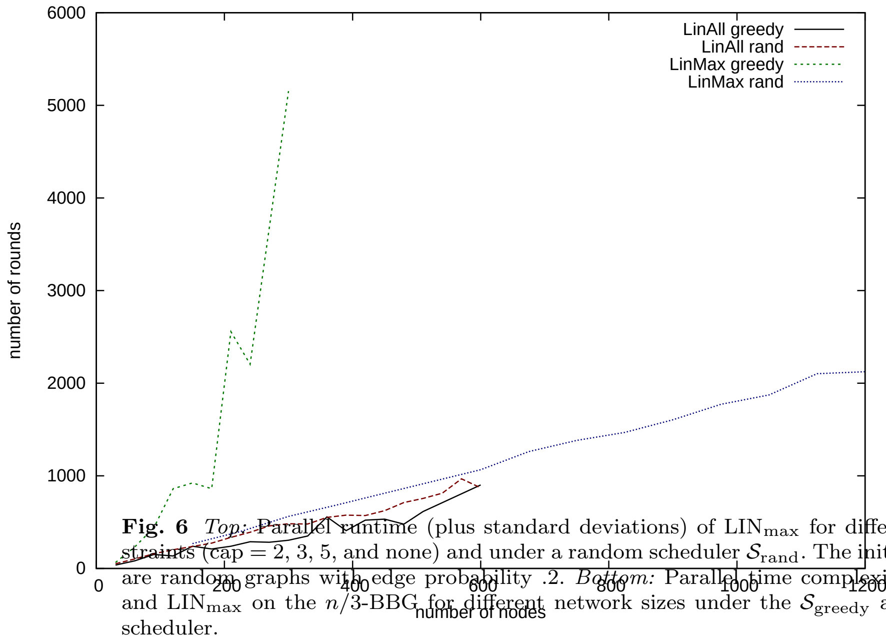


Fig. 6 *Top:* Parallel runtime (plus standard deviations) of LIN_{max} for different cap constraints (cap = 2, 3, 5, and none) and under a random scheduler $\mathcal{S}_{\text{rand}}$. The initial topologies are random graphs with edge probability .2. *Bottom:* Parallel time complexities of LIN_{all} and LIN_{max} on the $n/3$-BBG for different network sizes under the $\mathcal{S}_{\text{greedy}}$ and the $\mathcal{S}_{\text{rand}}$ scheduler.

canceled), however, for a model for concurrent executions of actions to be scalable, only a bounded number of executions of actions should be allowed to overlap at any node at any time. In order to come up with a simple and general model taking this into account, we decided to constrain the scheduling layer to independent sets of actions in each round.

An alternative model to study parallel complexity is the *worst-case critical path* [5,6] (i.e., the longest possible sequence of action executions that depend on each other) of a distributed execution of our linearization approach. However, it turns out that one can identify critical paths of length up to $\Theta(n^3)$ for our linearization approach, which is so far away from its real performance that the critical path notion seems to be too conservative and not meaningful in our context.

The critical path model can be defined in our framework in the following way. Consider a worst case scheduler that schedules one action (triple) per round. These triples form the nodes of a *directed acyclic graph* (DAG). An edge from an action A to a later action B is present, if and only if a connection that B requires to be present (or absent) was created (deleted) by A.

A simple graph family where the differences of the models become clear are the k-BBG graphs (cf Section 4). In the critical path model, LIN_{all} needs $\Theta(n^3)$ rounds to linearize the $n/3$-BBG, while in our model, LIN_{all} needs at most $O(n^2 \log n)$ rounds in the worst case (cf Theorem 3). In the following, we will show a lower bound for LIN_{all} on the $n/3$-BBG.

Theorem 7 *There is a graph, where a worst case scheduler $\mathcal{S}_{wc}$ for LIN_{all} needs time $\Omega(n^2)$ to finish.*

Proof Consider the following graph: the (even) nodes $v_2, v_4, \ldots, v_{k-2}, v_k$ have all edges to the nodes $v_{2k+1}, \ldots, v_{3k}$. A worst case scheduler can transform this graph in k rounds of LIN_{all} into the graph where the (odd) nodes $v_3, v_5, \ldots, v_{k-1}, v_{k+1}$ have all edges to the nodes $v_{2k+1}, \ldots, v_{3k}$: In the first round, node triple (v_2, v_3, v_{2k+1}) "moves" the "first" edge of node v_2 to node v_3, simultaneously with (v_4, v_5, v_{2k+2}) and so on, the i^{th} even node "moving" its i^{th} edge. More generally, in the j^{th} round, the i^{th} even node "moves" its $(i+j \mod k)^{th}$ edge to its right odd neighbor. After k such rounds the above described second graph is reached. The actions of one round form a maximal independent set because all long edges end at positions $v_2, \ldots, v_{k+1}$, and are hence blocked by one of the described triples.

In total, a worst case scheduler can perform the above k rounds k times by exchanging odd for even and shifting the left side further to the right. Additionally it uses a left shifted version of the above k rounds to transform the $n/3$ bipartite backbone graph into the described initial graph. □

Figure 6 (bottom) plots the performance of LIN_{all} and LIN_{max} on the BBG under different schedulers. Unfortunately, as some simulations require much computing resources, we have only generated experimental data up to certain network sizes. However, we can already see that while LIN_{all} is slow under $\mathcal{S}_{\text{greedy}}$, the other times are comparable and roughly linear.

For the critical path model, the picture looks quite different.

Theorem 8 *Under the critical path model, LIN_{all} needs time $\Theta(n^3)$ for the $n/3$-BBG.*

Proof Note that a single long edge $\{v_1, v_n\}$ will take $n-1$ linearization steps using backbone edges—edges between consecutive (w.r.t. IDs) nodes—before it is deleted. There are many such reduction sequences, one of them has as a last edge $\{v_1, v_3\}$, another one has $\{v_{n-2}, v_n\}$. The gist of the construction is to force all long edges to be deleted in this way at least between v_k and v_{2k+1}, and to make all these sequences depend on each other to form a long critical path.

More precisely, consider the long edges in order of increasing length (and for example increasing left endpoint). In this order, the edges get alternating colors *red* and *blue*. The semantics of the colors is that red edges are reduced to $\{v_k, v_{k+2}\}$, whereas blue edges are reduced to $\{v_{2k-1}, v_{2k+1}\}$ before they get deleted in one step.

Every edge is first changed to $\{v_k, v_{2k+1}\}$ using the backbone (these actions will not be part of the critical path). Because there are no shorter long edges, the edge does not become parallel to another long edge (which would mean it gets deleted). Then a red edge is reduced to $\{v_k, v_{2k-1}\}$ using the previous blue edge, and then this blue edge is deleted. Similarly, a blue edge is reduced to $\{v_{k+2}, v_{2k+1}\}$ using the previous red edge, and then this red edge is deleted. Then, in $k-3$ steps, a red edge is reduced to $\{v_k, v_{k+2}\}$, a blue edges to $\{v_{2k-1}, v_{2k+1}\}$.

In the critical path model, the shrinking of one edge along the middle part of the backbone depends on the previous edge already being reduced to an edge of length two. In total, this yields a sub-path of length $k-3$ on the critical path for every long edge, i.e., a critical path of length $k^2(k-3) \in \Theta(n^3)$. □

Finally, it remains to mention that in the model studied in [23], an adapted version of LIN_{all} would reduce the number of links in the $n/3$-BBG network from $\Theta(n^2)$ to $O(n)$ in only three rounds—performing a linear work per node and round, and thus ignoring the large contention. Subsequently, the linearization process requires a linear number of rounds until the graph is completely linearized. We believe that this behavior is not intuitive and that the insights that can be obtained with this model are limited.

6 Conclusion

This article has investigated the parallel complexity of self-stabilizing graph linearization. We have proposed a new model which we believe is more appropriate and intuitive than existing frameworks, and we provided a first analysis of the parallel time complexity of two most simple and archetypical self-stabilizing algorithms. We also conducted simulations of the algorithms proposed to complement our formal insights, and our experimental results indicate that our upper bounds may be too pessimistic.

We consider this work as a first step, and hope that our model will spark discussions and future research in the community. Indeed, we have started ourselves to consider 2-dimensional linearization problems [20] as well as scalable skip graphs [19]. However, both results are based on a simpler execution model that ignores node congestions. Moreover, it turns out that the 2-dimensional constructions require geometric reasoning that renders the analysis more complex, and it remains an open question how to apply our parallel runtime model in these more difficult settings.

References

1. D. Angluin, J. Aspnes, J. Chen, Y. Wu, and Y. Yin. Fast construction of overlay networks. In *Proc. of the 17th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 145–154, 2005.
2. J. Aspnes and G. Shah. Skip graphs. In *Proc. of the 14th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 384–393, 2003.
3. J. Aspnes and Y. Wu. $O(\log n)$-time overlay network construction from graphs with out-degree 1. In *Proc. of the 11th Int. Conference on Principles of Distributed Systems (OPODIS)*, volume 4878 of *LNCS*, pages 286–300. Springer, 2007.
4. B. Awerbuch and G. Varghese. Distributed program checking: A paradigm for building self-stabilizing distributed protocols. In *Proc. of the 32nd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 258–267, 1991.
5. R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, Feb. 1998.
6. R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, Sept. 1999.
7. J. Brzeziński and M. Szychowiak. Self-stabilization in distributed systems – a short survey. *Foundations of Computing and Decision Sciences*, 25(1):3–22, 2000.
8. I. Cidon, I. Gopal, and S. Kutten. New models and algorithms for future networks. *IEEE Transactions on Information Theory*, 41(3):769–780, May 1995.
9. T. Clouser, M. Nesterenko, and C. Scheideler. Tiara: A self-stabilizing deterministic skip list and skip graph. *Theoretical Computer Science*, 428:18–35, Apr. 2012.
10. C. Cramer and T. Fuhrmann. Self-stabilizing ring networks on connected graphs. Technical Report 2005-5, System Architecture Group, University of Karlsruhe, 2005.
11. E. W. Dijkstra. Self-stabilization in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.
12. D. Dolev, E. N. Hoch, and R. van Renesse. Self-stabilizing and Byzantine-tolerant overlay network. In *Proc. of the 11th Int. Conference on Principles of Distributed Systems (OPODIS)*, volume 4878 of *LNCS*, pages 343–357. Springer, 2007.
13. S. Dolev. *Self-Stabilization.* MIT Press, 2000.
14. S. Dolev and R. I. Kat. HyperTree for self-stabilizing peer-to-peer systems. *Distributed Computing*, 20(5):375–388, 2008.
15. D. Gall, R. Jacob, A. Richa, C. Scheideler, S. Schmid, and H. Täubig. Time complexity of distributed topological self-stabilization: The case of graph linearization. In *Proc. of the 9th Latin American Theoretical Informatics Symposium (LATIN)*, volume 6034 of *LNCS*, pages 294–305. Springer, 2010.
16. M. T. Goodrich and S. R. Kosaraju. Sorting on a parallel pointer machine with applications to set expression evaluation. *Journal of the ACM*, 43(2):331–361, 1996.
17. M. Harchol-Balter, T. Leighton, and D. Lewin. Resource discovery in distributed networks. In *Proc. of the 18th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 229–237, 1999.
18. T. Herman. Self-stabilization bibliography: Access guide. University of Iowa, 2002. See ftp://ftp.cs.uiowa.edu/pub/selfstab/bibliography/.

19. R. Jacob, A. Richa, C. Scheideler, S. Schmid, and H. Täubig. A distributed polylogarithmic time algorithm for self-stabilizing skip graphs. In *Proc. of the 28th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 131–140, 2009.
20. R. Jacob, S. Ritscher, C. Scheideler, and S. Schmid. Towards higher-dimensional topological self-stabilization: A distributed algorithm for Delaunay graphs. *Theoretical Computer Science*, 457:137–148, Oct. 2012.
21. S. Kniesburges, A. Koutsopoulos, and C. Scheideler. Re-Chord: A self-stabilizing Chord overlay network. In *Proc. of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 235–244, 2011.
22. T. Moscibroda, S. Schmid, and R. Wattenhofer. On the topologies formed by selfish peers. In *Proc. of the 25th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 133–142, 2006.
23. M. Onus, A. Richa, and C. Scheideler. Linearization: Locally self-stabilizing sorting in graphs. In *Proc. of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 99–108. SIAM, 2007.
24. A. Shaker and D. S. Reeves. Self-stabilizing structured ring topology P2P systems. In *Proc. of the 5th IEEE Int. Conference on Peer-to-Peer Computing (P2P)*, pages 39–46, 2005.
25. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. Technical Report MIT-LCS-TR-819, MIT, 2001.