T U M

I N S T I T U T F Ü R I N F O R M A T I K

Modeling Scalability in Distributed Self-Stabilization: The Case of Graph Linearization

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

TUM-I0835
November 08

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-11-I0835-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Modeling Scalability in Distributed Self-Stabilization: The Case of Graph Linearization

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

Abstract

This paper investigates how to efficiently and locally linearize graphs—i.e., how to build a sorted list of the nodes of a connected graph—in a distributed and self-stabilizing manner. This problem has many interesting application domains; for instance, self-stabilizing algorithms for graph linearization can serve as a building block to construct robust peer-to-peer overlays. A foremost question addressed in this paper is how to measure the efficiency of a given algorithm. We introduce a new model that takes into account the parallel complexity of a protocol. Our model avoids the scalability problems and bottlenecks of existing frameworks. We also 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, as well as their performance under a greedy selection of the actions to be executed. In particular, we show that one of the proposed algorithms achieves near-optimal parallel time complexity under such a greedy selection. We validate the behavior of these algorithms by experiments which confirm our formal findings and indicate that the runtimes may in fact be better in practice.

1 Introduction

Peer-to-peer systems such as open collaborative systems are becoming more and more popular these days. These systems are based on so-called overlay networks. In a broad sense, an overlay network is an application-based network formed by its participants on top of some physical network. In contrast to wired networks, the topology of an overlay network can freely and easily be changed by the participants. Such changes are also frequently necessary, for example because nodes enter or leave the system voluntarily, or because of a failure in a node or in the underlying network. Faults and inconsistencies should be regarded as the norm rather than an exception in overlay networks, mandating the need for highly efficient distributed update mechanisms to preserve desirable network properties. Minimum requirements for distributed overlay network protocols to be useful in practice are that they be local, simple, and self-stabilizing. Locality is important for fast distributed response times and for minimizing the impact of topology changes on the overlay network properties; simplicity is important

*Institut für Informatik, Technische Universität München, D-85748 Garching, Germany, dominik.gall@mytum.de, {jacob,scheidel,schmiste, taeubig}@in.tum.de

†Department of Computer Science and Engineering, Arizona State University, Tempe, AZ 85287-8809, USA, aricha@asu.edu

so that the protocols can be formally verified of their effectiveness; and self-stabilization is important for automatic recovery from any illegal state since protocols requiring centralized intervention will not necessarily scale to systems potentially spanning millions of sites.

In practice, many peer-to-peer systems—such as KaZaA, Bittorrent, and Kademlia—use heuristic methods in order to maintain their topology. They seem to recover well from degraded states though it is difficult to analyze that formally. Solutions presented in research publications mainly focus on maintaining scalable and well-structured overlay networks in an efficient manner [ABKM01, AS03, AS04, BKR+04, DR01, HJS+03, MNR02, RFH+01, SMK+01] but do not say much about how to recover these from a degraded state. For overlay networks that are based on a sorted list or ring (e.g., [AS03, AS04, HJS+03, SMK+01]), recovery mechanisms have been proposed as long as the base structure does not deviate significantly from a sorted list/ring (see Section 1.1). However, no efficient local mechanisms have been proposed for recovering a network once the base structure is significantly altered.

In this paper, we investigate how to recover a sorted list—i.e., how to *linearize* the graph—from *any* connected state. A first and foremost question in this context is how to model or measure the efficiency of a given distributed self-stabilizing algorithm. While researchers have proposed several solutions 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. This leads to a new family of execution models that distinguish actions proposed by an algorithm and a more or less adversarial scheduler that selects some of them for parallel execution. We focus on a very simple and natural linearization algorithm, such that the influence of the modeling becomes clear. We also consider a modified linearization algorithm that proposes fewer, but perhaps better, actions to the scheduler.

We specifically aim at exploring the parallel limitations (e.g., worst-case and best-case behavior) of the simple linearization algorithms proposed. For that we will assume the existence of some hypothetical schedulers. In particular, we consider a scheduler that always makes the worst possible, one that always makes the best possible, one that makes a random, and one that makes a "greedy" selection of actions to execute at any time step. 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. (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.)

1.1 Related Work

The first paper to study self-stabilization in the context of distributed computing was [Dij74] 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 et al.[AV91] 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 [BS00, Dol00, Her02].

Our paper 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 [DHvR07, DK08, SR05]. In the technical report of the distributed hash table Chord [SMK+01], 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 [AS03] 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 Protocol* [CF05] and the *Ring Network* [SR05] organize the nodes in a sorted ring. Unfortunately, both protocols have a large runtime. In [AAC+05], Aspnes 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 [AW07], a self-stabilizing algorithm is given which assumes that nodes initially have out-degree 1.

The papers closest to ours are by Onus et al. [ORS07] and by Clouser et al. [CNS08]. In [ORS07], a local-control strategy called *linearization* is presented for converting an arbitrary connected graph into a sorted list. However, it is only studied in a synchronous environment, and the strategy does not scale since in one time round it allows a node to communicate with an arbitrary number of its neighbors (which can be as high as $\Theta(n)$ for n nodes). Clouser et al. [CNS08] 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.

1.2 Our Contributions

The main contributions of this paper 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 [ORS07] and also gets rid of the scalability problems in [CNS08, ORS07] therefore allowing us to study the parallel time complexity of proposed linearization approaches. 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, as well their performances under a random and a greedy selection of the actions to be executed. In particular, we show that one of the proposed algorithms achieves near-optimal parallel time complexity under such a greedy selection. We also validate the behavior of these algorithms by experiments which not only confirm our formal findings, but indicate that the runtimes may in fact be better in practice. Finally, this paper discusses a particular situation that illustrates how the new model compares to others proposed in the literature.

1.3 Paper Organization

The remainder of this paper is organized as follows. In Section 2, we describe our model 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 a discussing our approach and comparing our model to alternative frameworks in Section 5, we conclude the paper 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) 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, $\mathrm{pred}(v)$ denotes the predecessor of v (i.e., the node $u \in V$ of largest identifier with $u < v$) and $\mathrm{succ}(v)$ denotes the successor of v according to "$<$". Two nodes u and v are called *consecutive* if and only if $u = \mathrm{succ}(v)$ or $v = \mathrm{succ}(u)$.

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

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*. An action has the form

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

where $<$ name $>$ is an *action label*, $<$ guard $>$ is a Boolean predicate over the (local and shared) variables of the executing node and $<$ 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. A scheduling layer is called *fair* if no action ever starves, i.e., if it is enabled for an unbounded amount of time, it will eventually be selected by the scheduler.

We model distributed computation as follows. The assignments of all local and shared variables defines a *system state*. Time proceeds in *rounds*. In each round, the scheduling layer may select any

set of independent actions to be executed by the nodes. The *work* performed in a round is equal to the number of actions selected by the scheduling layer in that round. A *computation* is a sequence of states such that for each state s_i at the beginning of round i, the next state s_{i+1} is obtained after executing all actions that were selected by the scheduling layer in round i. A distributed algorithm is called *self-stabilizing* w.r.t. a set of system states S and a set of *legal* states $L \subseteq S$ if for any initial state $s_1 \in S$ and any fair scheduling layer, the algorithm eventually arrives at a state $s \in L$.

Notice that this 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 state 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 paper.

2.1 Linearization

In this paper we are interested in designing distributed algorithms that can transform any initial graph into a sorted list (according to the node identifiers) using only local interactions between the nodes. A distributed algorithm is called *self-stabilizing* in this context if for any initial state that forms a connected graph, it eventually arrives at a state in which for all node pairs (u, v),

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

i.e., the nodes indeed form a sorted list. Once it arrives at this state, it should stay there, i.e., the state is a *fixpoint* of the algorithm. In the distributed algorithms studied in this paper, each node $u \in V$ repeatedly performs simple linearization steps in order to arrive at that fixpoint.

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

Figure 1: Left and right linearization step.

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.

1. Worst-case scheduler $\mathcal{S}_{\text{wc}}$: 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 $\mathcal{S}_{\text{rand}}$: 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 $\mathcal{S}_{\text{greedy}}$: 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 paper 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 $\mathcal{S}_{\text{opt}}$: 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 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 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_{all} and LIN_{max}

We first describe LIN_{all}. Algorithm LIN_{all} is very simple. Each node constantly tries to linearize its neighbors according to the *linearize left* and *linearize right* rules in Figure 1. In doing so, *all* possible triples on both sides are proposed to the scheduler. More formally, in LIN_{all} every node u checks the following actions for every pair of neighbors v and w:

$$\textbf{linearize left}(v,w): \quad (v,w \in u.L \ \wedge \ w < v < u) \ \rightarrow \ e(u,w) := 0, e(v,w) := 1$$

linearize right(v, w) : $(v, w \in u.R \ \wedge \ u < v < w) \ \rightarrow \ e(u, w) := 0, e(v, w) := 1$

LIN_{max} is similar to LIN_{all}: instead of proposing all possible triples on each side, LIN_{max} only proposes the triple which is the furthest on the corresponding side. Concretely, every node $u \in V$ checks the following actions 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 that these algorithms are correct in the sense that eventually, a linearized graph will be output.

Theorem 3.1. *LIN_{all} and LIN_{max} are self-stabilizing and converge to the linearized fixpoint.*

Proof. We first show that a linearization step of LIN_{all} and LIN_{max} cannot disconnect a connected 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 $\{v, w\}$, and the claim follows.

It remains to prove convergence to a line topology. 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., $\Psi = \sum_{e \in E} \text{len}(e)$. A linearization step reduces the potential Ψ by at least the length of the shorter edge in the triple, i.e., by $\text{len}(\{u, v\}) \geq 1$. Thus, if there is a single topology having a minimum value for potential Ψ, then this topology will eventually be reached by linearizing appropriate triples. The only topology respecting connectivity with minimum Ψ is the desired line topology. In any other connected state there must exist a triple (u, v, w) that allows a linearization step (which strictly reduces the value of Ψ). Therefore, the network converges to a line in a finite number of rounds, 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 3.2. *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:** 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 $\zeta_l(v) = 1/2$, and similarly for the right. We consider the potential function Φ which is defined as

$$\Phi = \sum_{v \in V} [(2\zeta_l(v) - 1) + (2\zeta_r(v) - 1)] = \sum_{v \in V} 2(\zeta_l(v) + \zeta_r(v) - 1)$$

Observe that initially, $\Phi_0 < 2n^2$, as $\zeta_l(v) + \zeta_r(v) < n$ for each node v. We show that after round i, the potential is at most $\Phi_i < 2n^2 - i$. Since $\text{LIN}_{\max}$ terminates (cf. also Theorem 3.1) with a potential $\Phi_j > 0$ for some j (the term of each node is positive, otherwise the node would be isolated), the claim follows. In order to see why the potential is reduced by at least one in every round, 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 distinguish two cases.

Case 1: Assume that $\{u, w\}$ was also the longest edge of w to the left. This implies that during linearization of the triple, we remove two longest edges (of nodes u and w) of length $\text{len}(\{u, w\})$ from the potential function. On the other hand, we may now have the following increase in the potential: u has a new longest edge $\{u, v\}$ to the right, v has a new longest edge $\{v, w\}$ to the right, and w has a new longest edge of length up to $\text{len}(\{u, w\}) - 1$ to the left. Summarizing, we get

$$\Delta\Phi \leq (2{\cdot}\text{len}(\{u, v\}){-}1){+}(2{\cdot}\text{len}(\{v, w\}){-}1){+}(2(\text{len}(\{u, w\}){-}1){-}1){-}(4 \cdot \text{len}(\{u, w\}) - 2) \leq -3$$

since $\text{len}(\{u, w\}) = \text{len}(\{u, v\}) + \text{len}(\{v, w\})$.

Case 2: Assume that $\{u, w\}$ was not the longest edge of w to the left. Then, by this linearization step, we remove edge $\{u, w\}$ from the potential function but may add edges $\{u, v\}$ (counted from node u to the right) and $\{v, w\}$ (counted from node v to the right). We have

$$\Delta\Phi \leq (2 \cdot \text{len}(\{u, v\}) - 1) + (2 \cdot \text{len}(\{v, w\}) - 1) - (2 \cdot \text{len}(\{u, w\}) - 1) \leq -1$$

since $\text{len}(\{u, w\}) = \text{len}(\{u, v\}) + \text{len}(\{v, w\})$. Since in every round, at least one triple can be linearized, 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, we will delete one link 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-ink 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 $\text{LIN}_{\max}$) 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 tipple 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 $n - 1$.

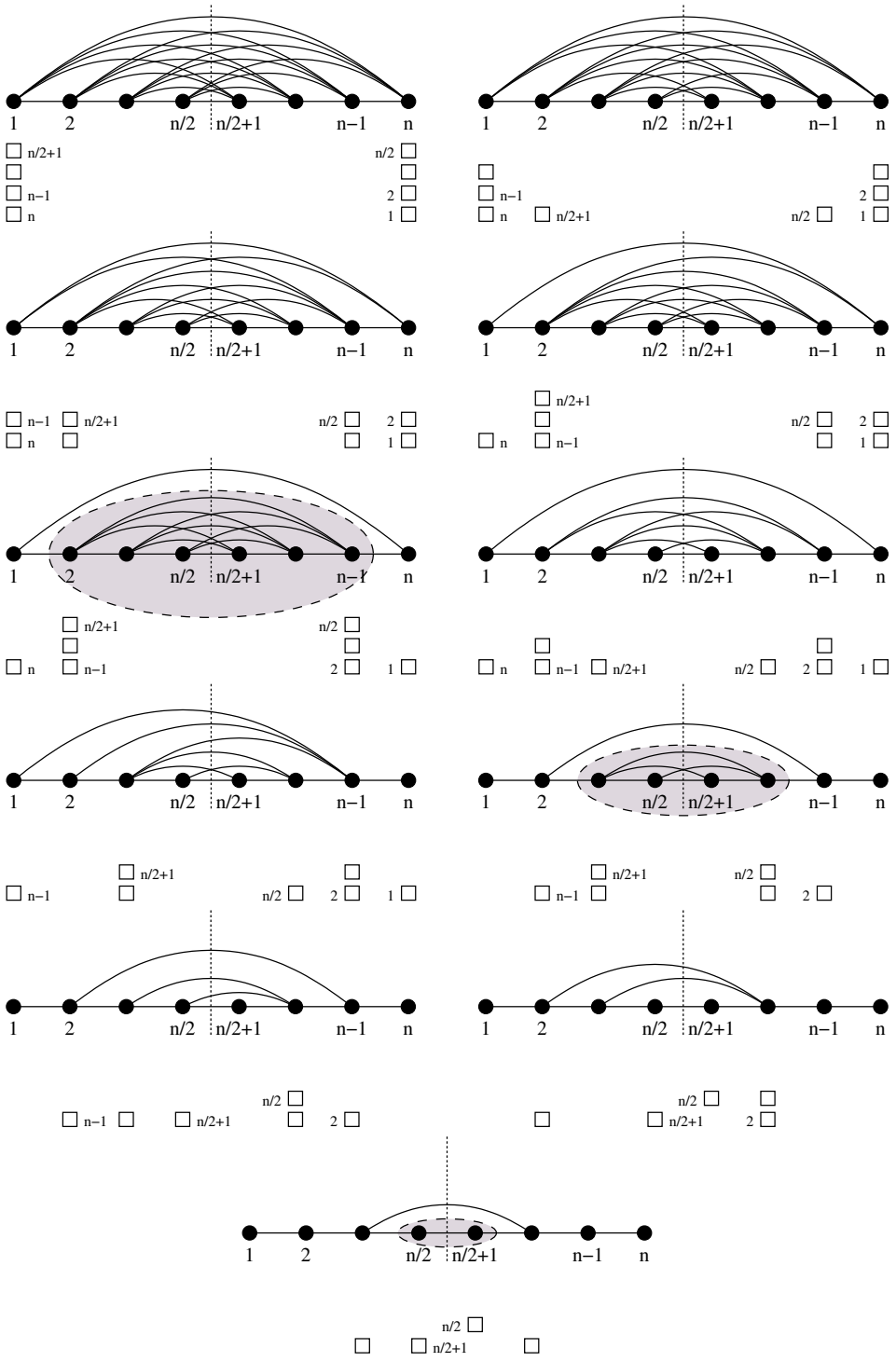

Figure 2: Bad case for linearizing a complete bipartite network.

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 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_{all} algorithm, we obtain a slightly higher upper bound. In the analysis, we need the following helper lemma.

Lemma 3.3. *Let Ξ be any positive potential function, where Ξ_0 is the initial potential value and Ξ_i is the potential after the i^{th} round of a given algorithm ALG. Assume that $\Xi_i \leq \Xi_{i-1}\cdot(1-1/f)$ and that ALG terminates if $\Xi_j \leq \Xi_{stop}$ for some $j \in \mathbb{N}$. Then, the runtime of ALG is at most $O(f\cdot\log(\Xi_0/\Xi_{stop}))$ rounds.*

Proof. From $\Xi_i \leq \Xi_{i-1}\cdot(1-1/f)$, it follows that $\Xi_j \leq \Xi_0\cdot(1-1/f)^j$.
Now consider $j = f\cdot\ln\frac{\Xi_0}{\Xi_{stop}}$, which leads to (using $\ln(1+x)\leq x$ for all $x>-1$)

$$\Xi_j \leq \Xi_0\cdot(1-1/f)^{f\cdot\ln\frac{\Xi_0}{\Xi_{stop}}} = \Xi_0 e^{f\cdot\left(\ln\frac{\Xi_{stop}}{\Xi_0}\right)\cdot\ln(1-1/f)} \leq \Xi_0 e^{f\cdot\left(\ln\frac{\Xi_0}{\Xi_{stop}}\right)\cdot(-1/f)} = \Xi_0 e^{-\ln\frac{\Xi_0}{\Xi_{stop}}} = \Xi_{stop}.$$

□

Theorem 3.4. *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 $\Psi = \sum_{e\in E}\text{len}(e)$, for which it holds that $\Psi_0 < n^3$. 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. (Left-linearizations are similar and not discussed further here.) During a linearization step, the sum of

the edge lengths is reduced by at least one. Similarly to the proof of Theorem 3.5, we want to calculate the amount of blocked potential in a round due to the linearization of the triple (u, v, w). Nodes u, v, and w have at most $\widehat{\deg}(u) + \widehat{\deg}(v) + \widehat{\deg}(w) < n$ many independent neighbors. In the worst case, when the triple's incident edges are removed (blocked potential 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 3.3, the claim follows. □

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) 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 3.5. *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. We consider the potential function

$$\Psi = \sum_{e \in E} \text{len}(e).$$

Initially, Ψ_0 must be smaller than $n^2(n-1)$, since there are less than n^2 edges of length at most $n-1$. 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 3.3 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 $\widehat{\deg}(v_1)/2 - 1 \geq \widehat{\deg}(v_1)/4$, 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—according to $\mathcal{S}_{\text{greedy}}$—$v_1$ has at least as many remaining neighbors on the right as it has on the left, we have that $\text{dist}(v_1, v_2) \geq \widehat{\deg}(v_1)/2 - 1 \geq \widehat{\deg}(v_1)/4$.

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 3.6. *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 $G_0 = (V, E)$ where $\forall i$ such that $0 < i < n-1$: $\{v_i, v_{i+1}\} \in E$. Additionally, E contains a long edge $e := \{v_1, v_n\} \in E$. 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 at most 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$_{\text{max}}$ 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 only one left neighbor, and during right-linearizations we allow a degree increase to the right if the middle node has degree one. In other words, we study a *degree cap of two*.

We find that both our algorithms LIN$_{\text{all}}$ and LIN$_{\text{max}}$ still terminate with a correct solution under this restrictive model.

Theorem 3.7. *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 3.2. We only need to prove additionally 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 $\Psi = \sum_{e \in E} \mathrm{len}(e)$ 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$_{\text{all}}$ and LIN$_{\max}$ is typically much better than shown in Theorem 3.7. Moreover, it turns out that even without imposing a degree cap, LIN$_{\text{all}}$ and LIN$_{\max}$ do not increase the maximal degrees during their computations *automatically*.

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.

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 $V = \{v_1, \ldots, v_n\}$. It has n nodes that are all 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*.

Many experiments have been conducted to shed light onto the parallel runtime of LIN_{all} and LIN_{max} in different networks. Figure 3 (left) 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 (right) 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.

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 (left) 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.

5 Discussion and Model Comparison

This section provides a short discussion of our model. We also compare our approach to alternative models, e.g., to the so-called *critical path model* introduced in [BL98, BL99].

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. It turns out that the independence requirement is not necessary in this particular case as it would be sufficient if each node initiates a new linearization only after its previously initiated linearization has been completed (or 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* [BL98, BL99] (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 is not meaningful in our context.

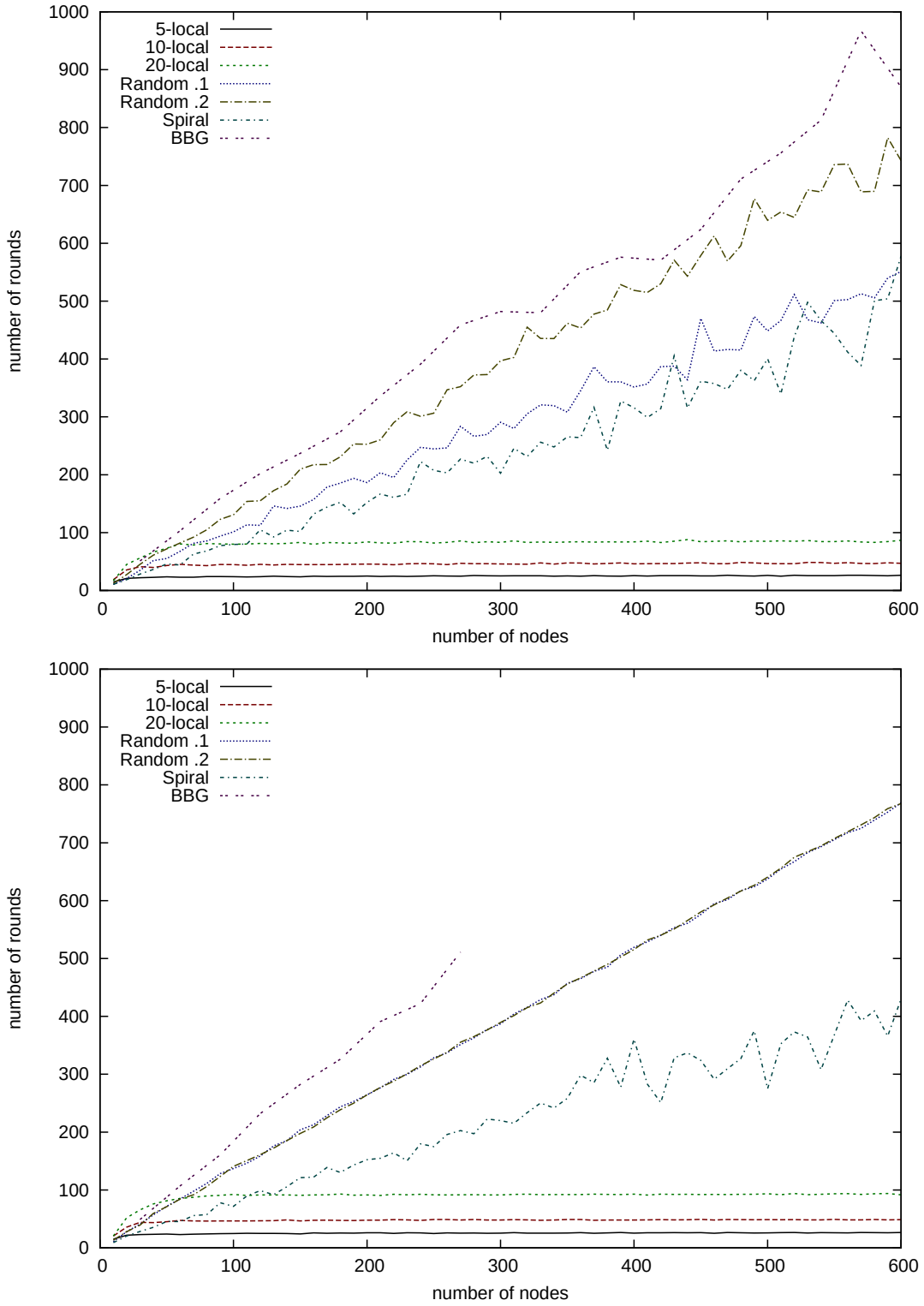

Figure 3: *Top:* Parallel runtime of LIN_{all} for different graphs under $\mathcal{S}_{\text{rand}}$: two k-local graphs with $k = 5$, $k = 10$ and $k = 20$, two random graphs with $p = .1$ and $p = .2$, a spiral graph and a $n/3$-BBG. *Bottom:* Same experiments with LIN_{max}.

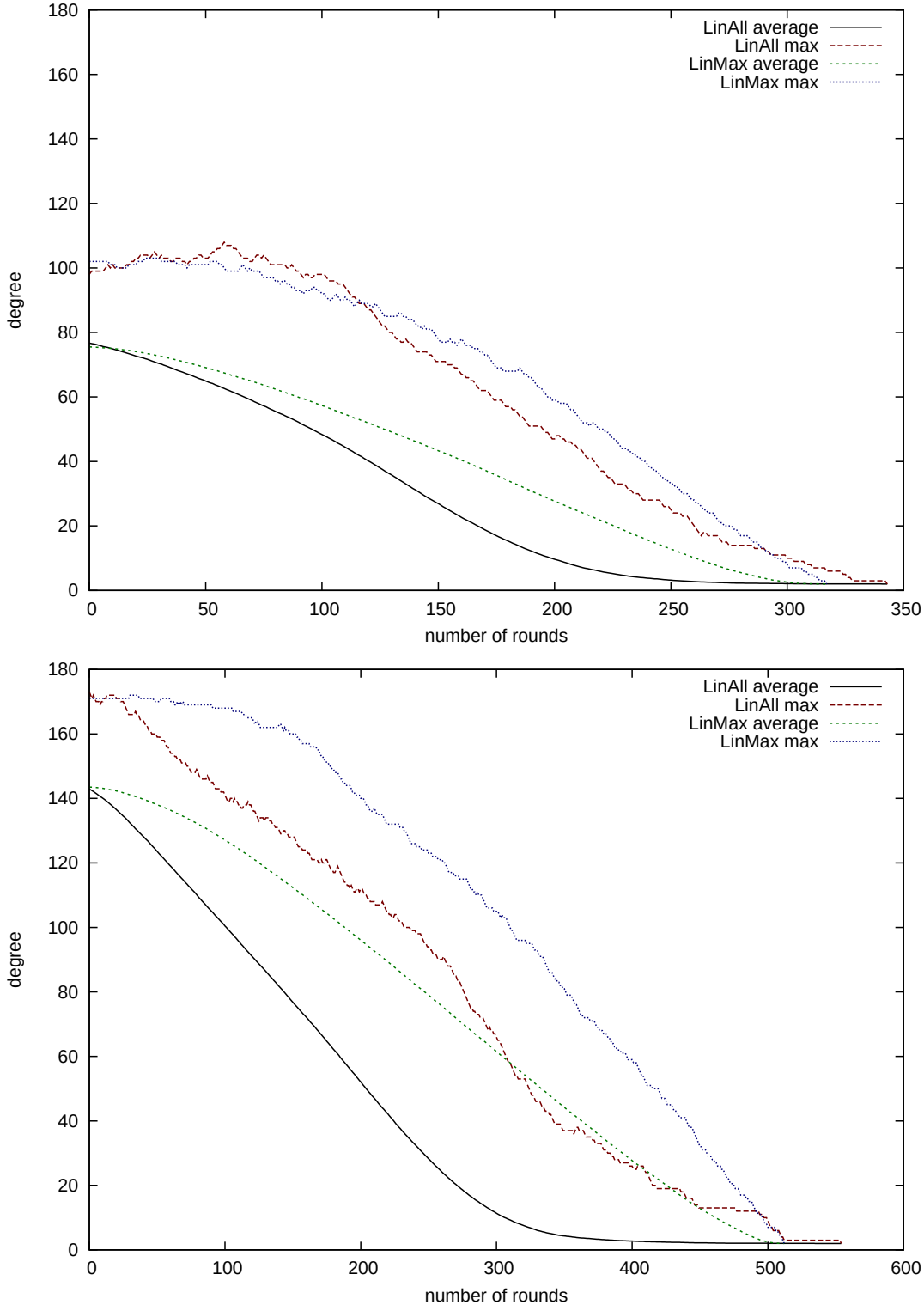

Figure 4: *Top:* Maximum and average degree during a run of LIN_{all} and LIN_{max} on a random graph with edge probability $p = .1$. *Bottom:* The same experiment on a random graph with $p = .2$.

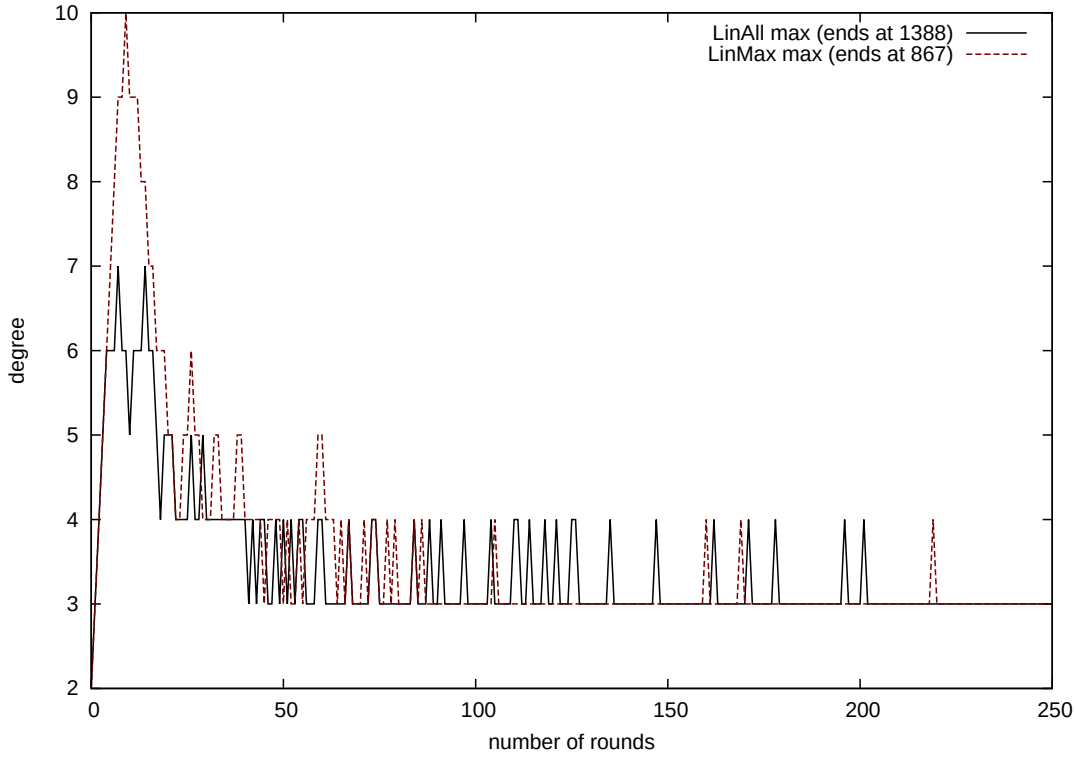

Figure 5: Evolution of maximal degree on spiral graphs under a randomized scheduler $\mathcal{S}_{\text{rand}}$.

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, iff 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 3.4). In the following, we will show a lower bound for LIN_{all} on the $n/3$-BBG.

Theorem 5.1. *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.

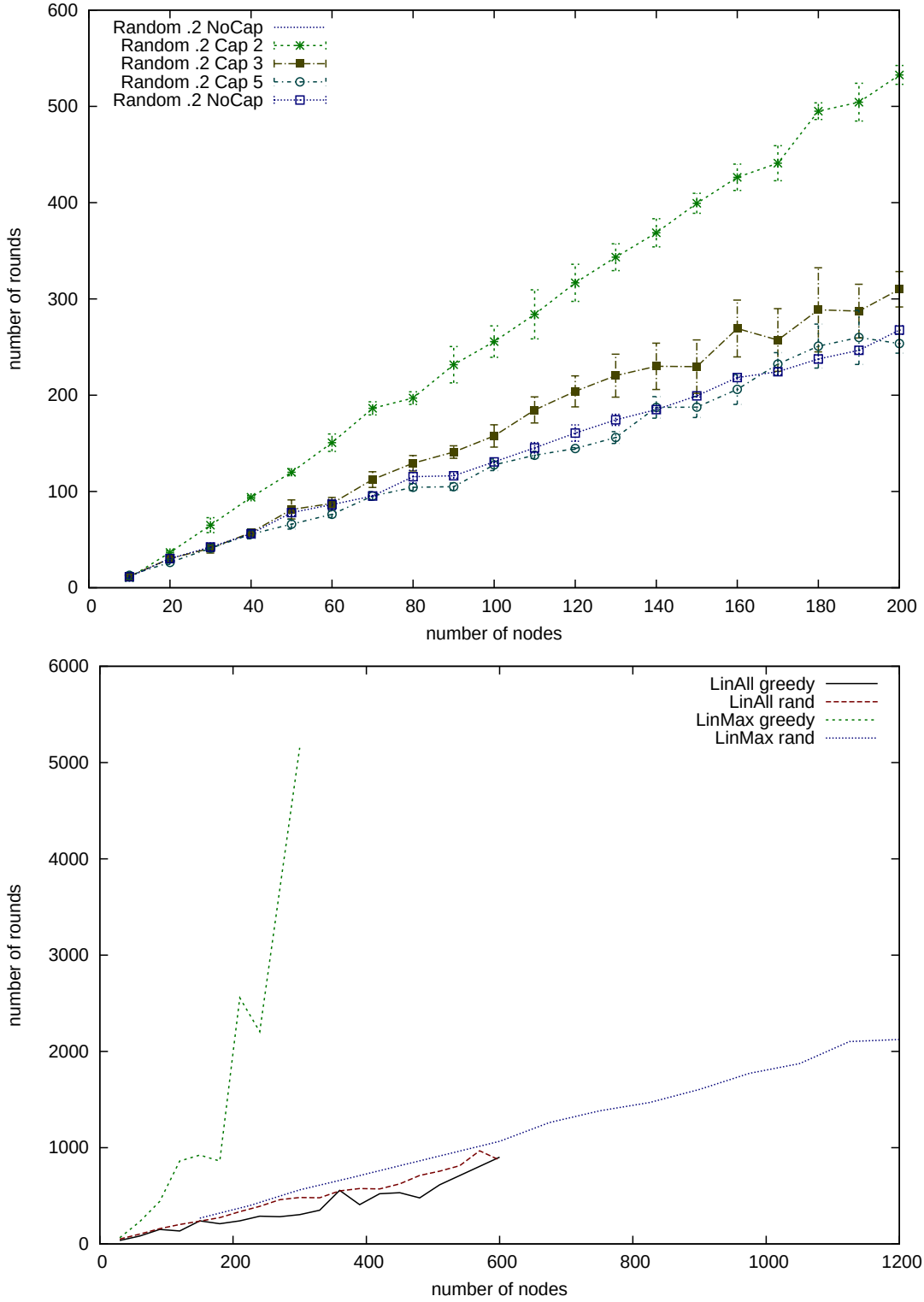

Figure 6: *Top:* Parallel time complexities (plus standard deviations) of LIN_{max} for different cap constraints (cap = 2, 3, 5, and none) and under a random scheduler $\mathcal{S}_{\text{rand}}$. The initial topologies are random graphs with edge probability .2. *Bottom:* Parallel time complexities 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.

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 (right) 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 5.2. *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 [ORS07], 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 competely linearized. We believe that this behavior is not intuitive and that the insights that can be obtained with this model are severely limited.

6 Conclusion

This paper has investigated the distributed 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 a simple self-stabilizing algorithm. We also conducted extensive simulations of the algorithms proposed: In fact, the experiments indicate that our upper bounds can be improved and may even match the lower bounds.

We consider this paper as a first step, and hope that our model will spark discussions and future research in the community. We have three different directions of impact on our own research agenda. First, we plan to analyze more sophisticated graph linearization algorithms, perhaps even pinpointing the complexity of the problem and not of an algorithm. Second, we seek further self-stabilizing construction of other topologies such as hypercube graphs or skip graphs. Finally, we want to port our model into the realm of local-control message-passing systems.

References

[AAC+05] Dana Angluin, James Aspnes, Jiang Chen, Yinghua Wu, and Yitong Yin. Fast construction of overlay networks. In *Proc. 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 145–154, 2005.

[ABKM01] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *Proc. 18th ACM Symposium on Operating Systems Principles(SOSP)*, pages 131–145, 2001.

[AS03] James Aspnes and Gauri Shah. Skip graphs. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 384–393, 2003.

[AS04] Baruch Awerbuch and Christian Scheideler. The hyperring: A low-congestion deterministic data structure for distributed environments. In *Proc. 15th Ann. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 318–327, 2004.

[AV91] Baruch Awerbuch and George Varghese. Distributed program checking: A paradigm for building self-stabilizing distributed protocols. In *Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 258–267, 1991.

[AW07] James Aspnes and Yinghua Wu. $O(\log n)$-time overlay network construction from graphs with out-degree 1. In *Proc. International Conference on Principles of Distributed Systems (OPODIS)*, volume 4878 of *LNCS*, pages 286–300, 2007.

[BKR+04] Ankur Bhargava, Kishore Kothapalli, Chris Riley, Christian Scheideler, and Mark Thober. Pagoda: A dynamic overlay network for routing, data management, and multicasting. In *Proc. 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 170–179, 2004.

[BL98] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, February 1998.

[BL99] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[BS00] Jerzy Brzeziński and Michał Szychowiak. Self-stabilization in distributed systems – a short survey. *Foundations of Computing and Decision Sciences*, 25(1), 2000.

[CF05] Curt Cramer and Thomas Fuhrmann. Self-stabilizing ring networks on connected graphs. Technical Report 2005-5, System Architecture Group, University of Karlsruhe, 2005.

[CNS08] Thomas Clouser, Mikhail Nesterenko, and Christian Scheideler. Tiara: A self-stabilizing deterministic skip list. In *Proc. 10th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2008.

[DHvR07] Danny Dolev, Ezra N. Hoch, and Robbert van Renesse. Self-stabilizing and byzantine-tolerant overlay network. In *Proc. International Conference on Principles of Distributed Systems (OPODIS)*, volume 4878 of *LNCS*, pages 343–357, 2007.

[Dij74] Edsger W. Dijkstra. Self-stabilization in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[DK08] Shlomi Dolev and Ronen I. Kat. Hypertree for self-stabilizing peer-to-peer systems. *Distributed Computing*, 20(5):375–388, 2008.

[Dol00] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.

[DR01] Peter Druschel and Antony Rowstron. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 2001. See also `http://research.microsoft.com/~antr/Pastry`.

[Her02] T. Herman. Self-stabilization bibliography: Access guide. University of Iowa, 2002.

[HJS+03] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proc. 4th USENIX Symposium on Internet Technologies and Systems (USITS)*, pages 113–126, 2003.

[MNR02] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proc. 21st Annual Symposium on Principles of Distributed Computing (PODC)*, pages 183–192, 2002.

[ORS07] Melih Onus, Andrea Richa, and Christian Scheideler. Linearization: Locally self-stabilizing sorting in graphs. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2007.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 161–172, 2001.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. Technical Report MIT-LCS-TR-819, MIT, 2001.

[SR05] Ayman Shaker and Douglas S. Reeves. Self-stabilizing structured ring topology P2P systems. In *Proc. 5th IEEE International Conference on Peer-to-Peer Computing*, pages 39–46, 2005.