Time Complexity of Distributed Topological Self-stabilization: The Case of Graph Linearization*

Dominik Gall[1], Riko Jacob[1], Andrea Richa[2], Christian Scheideler[3], Stefan Schmid[4], and Hanjo Täubig[1]

[1] Institut für Informatik, TU München, Garching, Germany
[2] Dept. Computer Science and Engineering, Arizona State University, Tempe, USA
[3] Dept. Computer Science, University of Paderborn, Paderborn, Germany
[4] TU Berlin / Deutsche Telekom Laboratories, Berlin, Germany

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 that converge quickly to such a desirable topology, independently of the initial network state. This paper 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 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. We propose two variants of a simple algorithm, and provide an extensive formal analysis 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.

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, when designing such a system, a prime concern is *robustness*. 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 self-stabilizing system guarantees that from *any* connected topology, eventually an overlay with desirable properties will result.

In this paper, 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

* For a complete technical report, we refer the reader to [8]. Research supported by the DFG project SCHE 1592/1-1, and NSF Award number CCF-0830704.

A. López-Ortiz (Ed.): LATIN 2010, LNCS 6034, pp. 294–305, 2010.
© Springer-Verlag Berlin Heidelberg 2010

connected network. We want to investigate how long it takes until the topology reaches a (to be specified) desirable state. 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 employ our tools to the problem of *graph linearization* where nodes—initially in an arbitrary connected graph—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 extensive analysis, graph linearization under our model is already non-trivial and reveals an interesting structure. This paper 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 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.[1]

1.1 Related Work

The construction and maintenance of a given network structure is of prime importance in many distributed systems, for example in peer-to-peer computing [6,7,11,13,14]; e.g., Kuhn et al. [11] showed how to maintain a hypercube under dynamic worst-case joins and leaves, and Scheideler et al. [13] presented the SHELL network which allows peers to join and leave efficiently. In the technical report of the distributed hash table Chord [15], stabilization protocols are described which allow the topology to recover from certain degenerate situations. Unfortunately, however, in these papers, no algorithms are given to recover from *arbitrary* states.

Also skip graphs [1] can be repaired from certain states, namely states which resulted from node faults and inconsistencies due to churn. In a recent paper, we have shown [9] that a modified and locally checkable version of the skip graph can be built from any connected network. Interestingly, the algorithm introduced in [9] self-stabilizes in polylogarithmic time. Unfortunately, however, while the resulting structure is indeed scalable, the number of edges can become quadratic

[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.

during the execution of that algorithm. Moreover, the execution model does not scale either.

In this paper, in order to be able to focus on the main phenomena of our model, we chose to restrict ourselves to topological *linearization*. Linearization and the related problem of organizing nodes in a ring is also subject to active research. The *Iterative Successor Pointer Rewiring Protocol* [5] and the *Ring Network* [14] organize the nodes in a sorted ring. Unfortunately, both protocols have a large runtime. We have recently started to work on 2-dimensional linearization problems for a different (classic) time-complexity model [10].

The papers closest to ours are by Onus et al. [12] and by Clouser et al. [4]. In [12], 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. [4] 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 contribution of this paper is two-fold. First, we present an alternative approach to modeling scalability of distributed, self-stabilizing algorithms that does not require synchronous executions like in [12] and also gets rid of the scalability problems in [4,12] 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, and also study their performance under a random and a greedy selection of the actions to be executed.

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, $\text{pred}(v)$ 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 $G = (V, E)$ 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 $e(u, v)$. (E.g., node u sends a change request message to u. 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 (with larger IDs) 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 $\text{dist}(u, v)$ 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*. 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 interactions a node is involved in (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 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 (and stays) at a state $s \in L$.

Fig. 1. Left and right linearization step

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 the (only) *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*.

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 an upper 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:

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_{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 (w.r.t. IDs) 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):
$(v, w \in u.L) \wedge w < v < u \wedge \nexists x \in u.L \backslash \{w\} : x < v) \rightarrow \ e(u, w) := 0, e(v, w) := 1$

linearize right(v, w):
$(v, w \in u.R) \wedge u < v < w \wedge \nexists x \in u.R \backslash \{w\} : x > v) \rightarrow \ e(u, w) := 0, e(v, w) := 1$

We first show that these algorithms are correct in the sense that eventually, a linearized graph will be output.

Theorem 1. *LIN_{all} and LIN_{max} are self-stabilizing and converge to the sorted list.*

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. Due to space constraints, we only give a proof sketch for the 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 LIN_{max} terminates (cf. also Theorem 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

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

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.

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

Lemma 1. *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.*

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 $\Psi = \sum_{e \in E} \text{len}(e)$, 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 4, 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 1, 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 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.*

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.*

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_{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 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_{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.*

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 (Erdös-Rényi graph), 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):* This seems to be a "hard" graph which also allows to compare different models. 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 2 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 2 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 3 (top) 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 3 have a high initial degree, and it is interesting to analyze what happens in case of sparse initial graphs. Figure 3

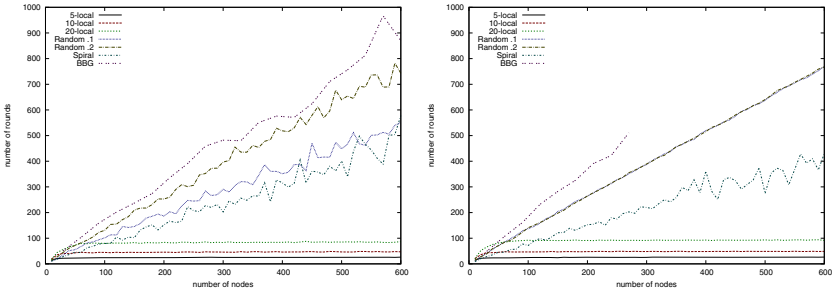

Fig. 2. *Left:* Parallel runtime of $\mathrm{LIN}_{\mathrm{all}}$ for different graphs under $\mathcal{S}_{\mathrm{rand}}$: two k-local graphs with $k = 5$, $k = 10$ and $k = 20$, two random graphs with $p = .1$ and $p = .2$, a spiral graph and a $n/3$-BBG. *Right:* Same experiments with $\mathrm{LIN}_{\mathrm{max}}$.

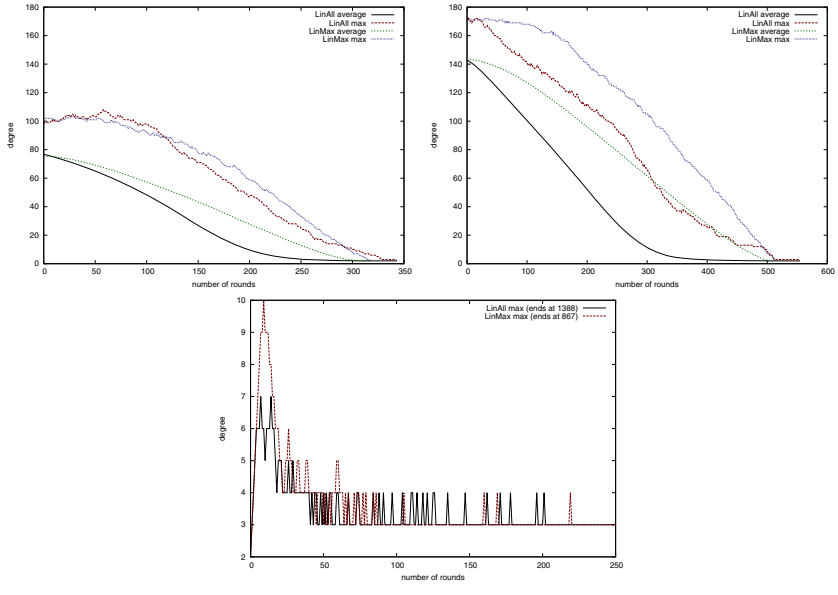

Fig. 3. *Top left:* Maximum and average degree during a run of $\mathrm{LIN}_{\mathrm{all}}$ and $\mathrm{LIN}_{\mathrm{max}}$ on a random graph with edge probability $p = .1$. *Top right:* The same experiment on a random graph with $p = .2$. *Bottom:* Evolution of maximal degree on spiral graphs under a randomized scheduler $\mathcal{S}_{\mathrm{rand}}$.

(bottom) 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.

References

1. Aspnes, J., Shah, G.: Skip graphs. In: Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 384–393 (2003)
2. Blumofe, R.D., Leiserson, C.E.: Space-e cient scheduling of multithreaded computations. SIAM Journal on Computing 27(1), 202–229 (1998)
3. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. Journal of the ACM 46(5), 720–748 (1999)
4. Clouser, T., Nesterenko, M., Scheideler, C.: Tiara: A self-stabilizing deterministic skip list. In: Kulkarni, S., Schiper, A. (eds.) SSS 2008. LNCS, vol. 5340, pp. 124–140. Springer, Heidelberg (2008)
5. Cramer, C., Fuhrmann, T.: Self-stabilizing ring networks on connected graphs. Technical Report 2005-5, System Architecture Group, University of Karlsruhe (2005)
6. Dolev, D., Hoch, E.N., van Renesse, R.: Self-stabilizing and byzantine-tolerant overlay network. In: Tovar, E., Tsigas, P., Fouchal, H. (eds.) OPODIS 2007. LNCS, vol. 4878, pp. 343–357. Springer, Heidelberg (2007)
7. Dolev, S., Kat, R.I.: Hypertree for self-stabilizing peer-to-peer systems. Distributed Computing 20(5), 375–388 (2008)
8. Gall, D., Jacob, R., Richa, A., Scheideler, C., Schmid, S., Täubig, H.: Modeling scalability in distributed self-stabilization: The case of graph linearization. Technical Report TUM-I0835, Technische Universität München, Computer Science Dept. (November 2008)
9. Jacob, R., Richa, A., Scheideler, C., Schmid, S., Täubig, H.: A distributed polylogarithmic time algorithm for self-stabilizing skip graphs. In: Proc. ACM Symp. on Principles of Distributed Computing, PODC (2009)
10. Jacob, R., Ritscher, S., Scheideler, C., Schmid, S.: A self-stabilizing and local delaunay graph construction. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878. Springer, Heidelberg (2009)
11. Kuhn, F., Schmid, S., Wattenhofer, R.: A self-repairing peer-to-peer system resilient to dynamic adversarial churn. In: Castro, M., van Renesse, R. (eds.) IPTPS 2005. LNCS, vol. 3640, pp. 13–23. Springer, Heidelberg (2005)
12. Onus, M., Richa, A., Scheideler, C.: Linearization: Locally self-stabilizing sorting in graphs. In: Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX). SIAM, Philadelphia (2007)
13. Scheideler, C., Schmid, S.: A distributed and oblivious heap. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5556, pp. 571–582. Springer, Heidelberg (2009)
14. Shaker, A., Reeves, D.S.: Self-stabilizing structured ring topology P2P systems. In: Proc. 5th IEEE International Conference on Peer-to-Peer Computing, pp. 39–46 (2005)
15. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. Technical Report MIT-LCS-TR-819. MIT (2001)