Adding Networks*

Panagiota Fatourou† *Maurice Herlihy*‡

Abstract

An *adding network* is a distributed data structure that supports a concurrent, lock-free, low-contention implementation of a fetch-and-add counter. (A counting network is an instance of an adding network that supports only fetch-and-increment.)

We give a lower bound showing that adding networks have inherently high latency. Any adding network powerful enough to support addition by at least two values a and b, where $|a| > |b| > 0$, has sequential executions in which each token traverses $\Omega(n/c)$ switching elements, where n is the number of concurrent processes, and c is a quantity we call *one-shot contention*. For many kinds of "reasonable" switching networks (including counting networks) one-shot contention is constant. (Counting networks, by contrast, have $O(\log n)$ latency [6, 11].)

This bound is tight. We give the first concurrent, lock-free, low-contention networked data structure that supports arbitrary fetch-and-add operations.

*__Not eligible for the best student paper award. Submitted to both regular and brief announcements tracks.__

†**Contact Author**. *Postal address*: MaxPlanck Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany. Part of the work of this author was performed while visiting the Department of Computer Science, Brown University, Providence, RI 02912, USA. E-mail: `faturu@mpi-sb.mpg.de`

‡Department of Computer Science, Brown University, Providence, RI 02912, USA. Email: `herlihy@cs.brown.edu`

1 Introduction

Motivation-Overview

A *fetch&increment* variable provides an operation that atomically adds one to its value and returns its prior value. Applications of *fetch&increment* counters include shared pools and stacks, load balancing, and software barriers.

A *counting network* [3] is a class of distributed data structures used to construct concurrent, low-contention implementations of *fetch&increment* counters. A limitation of the original counting network constructions is that the resulting shared counters can be incremented, but not decremented. More recently, Shavit and Touitou [16] showed how to extend certain counting network constructions to support decrement operations, and Aiello and others [2] extended this technique to arbitrary counting network constructions.

In this paper we consider the following natural generalization of these recent results: can we construct network data structures that support lock-free, highly-concurrent, low-contention *fetch&add* operations? (Recall that *fetch&add* atomically adds an arbitrary value to a shared variable, and returns the variable's prior value.)

We address these problems in the context of concurrent *switching networks*, a generalization of the balancing networks used to construct counting networks. As discussed in more detail below, a switching network is a directed graph, where edges are called *wires* and node are called *switches*. Each process shepherds a *token* through the network. Switches and tokens are allowed to have internal states. A token arrives at a switch via an input wire. In one atomic step, the switch absorbs the token, changes its state and possibly the token's state, and emits the token on an output wire.

Let S be any non-empty set of integer values. An *S-adding network* is a switching network that implements the set of operations `fetch&add`$(\cdot,s)$, for s an element of S. As a special case, an (a,b)-adding network supports two operations: `fetch&add`$(\cdot,a)$ and `fetch&add`$(\cdot,b)$. A process executes a `fetch&add`$(\cdot,a)$ operation by shepherding a token of *weight* a through the network.

Our results encompass both bad news and good news. First the bad news. We define the network's *one-shot contention* to be the largest number of tokens that can meet at a single switch in an execution in which the tokens enter on distinct wires. For counting networks, and perhaps for "reasonable" switching networks, this quantity is constant. We show that for any (a,b)-adding network, where $|a| > |b| > 0$, there exist n-process sequential executions where each process traverses $\Omega(n/c)$ switches, where c is the network's one-shot contention. This result implies that any lock-free low-contention adding network must have high worst-case latency, even in the absence of concurrency. As an aside, we note that there are two interesting cases not subject to our lower bound: a low-latency $(a,-a)$-adding network is given by the antitoken construction, and an $(a,0)$-adding network is just a regular counting network augmented by a pure read operation.

Now for the good news. We introduce a novel construction for a lock-free, low-contention *fetch&add* switching network, called `Ladder`, in which processes take $O(n)$ steps on average. Tokens carry mutable values, and switching elements are balancers augmented by atomic read-

write variables. The construction is lock-free, but not wait-free (meaning that individual tokens can be overtaken arbitrarily often, but that some tokens will always emerge from the network in a finite number of steps).

An ideal *fetch&add* switching network is (1) lock-free, with (2) low contention, and (3) low latency. Although this paper shows that no switching network can have all three properties, any two are possible:* a single switch is lock-free with low latency, but has high contention, a combining network [8, 9] has low contention and $O(\log n)$ latency but requires tokens to wait for one another, and the construction presented here is lock-free with low contention, but has $O(n)$ latency.

Related Work

Counting networks were first introduced in by Aspnes *et. al* [3]. A flurry of research on counting networks followed (see e.g., [1, 2, 4, 5, 6, 7, 10, 11, 14, 16, 17]). Counting networks are limited to support only *fetch&increment* and *fetch&decrement* operations. Our work is the first to study whether lock-free network data structures can support even more complex operations. We generalize traditional counting networks by introducing switching networks, which employ more powerful switches. Switches can be shared objects characterized by arbitrary internal states. Moreover, each token is allowed to have a state by maintaining its own variables; tokens can exchange information with the switches they traverse.

Surprisingly, it turns out that supporting even the slightly more complex operation of *fetch&add*, where adding is by only two different integers a, b such that $|a| > |b| > 0$, is as difficult as ensuring linearizability [10]. In [10] the authors prove that there exists no ideal linearizable counting network. In a corresponding way, our lower bound implies that even the most powerful switching networks cannot guarantee efficient support of this relatively simple *fetch&add* operation.

The LADDER switching network has the same topology as the linearizable SKEW presented by Herlihy and others [10], but the behavior of the LADDER network is significantly different. In this network, tokens accumulate state as they traverse the network, and they use that state to determine how they interact with switches. The resulting network is substantially more powerful, and requires a substantially different analysis.

Organization

This paper is organized as follows. Section 2 introduces switching networks. Our lower bound is presented in Section 3, while the `Ladder` network is presented and analyzed in Section 4. Due to lack of space, most of our proofs are only sketched in this paper.

*NASA's motto "faster, cheaper, better" has been satirized as "faster, cheaper, better: pick any two".

2 Switching Networks

A *switching network*, like a counting network [3], is a directed graph whose nodes are simple computing elements called *switches*, and whose edges are called *wires*. A wire directed from switch s to switch s' is an *output wire* for s and an *input wire* for s'. Each *token* (input item) enters on one of the network's w_{in} input wires, traverses a sequence of switches, and leaves on one of the network's w_{out} output wires. A (w_{in}, w_{out})-switching network has w_{in} input wires and w_{out} output wires. For each input index i, $0 \leq i \leq w_{in} - 1$, we denote by X_i the number of tokens that have entered on the network's input wire i and for each output index j, $0 \leq j \leq w_{out} - 1$, we denote by Y_j the number of tokens that have exited on the network's output wire j.

A *switch* is a shared data object characterized by an internal state, its set of f_{in} *input wires*, labeled $0, \ldots, f_{in} - 1$, and f_{out} *output wires*, labeled $0, \ldots, f_{out} - 1$. (The values f_{in} and f_{out} are called the switch's *fan-in* and *fan-out* respectively).

There are n processes that shepherd *tokens* through the network. A process shepherds only one token at a time, but it can start shepherding a new token as soon as its previous token has emerged from the network. In a departure from counting networks, a token has a mutable state, which can change as it traverses the network.

A switch acts as a router for tokens. When a token arrives on a switch's input wire, the following events occur atomically: (1) the switch removes the token from the input wire, (2) the switch changes state, (3) the token changes state, and (4) the switch places the token on an output wire. Tokens are asynchronous, but they do not fail. The wires are one-way communication channels. Communication is asynchronous, unordered, but reliable (meaning a token does not wait on a wire forever). For example, a (k, ℓ)-*balancer* is a switch with fan-in k and fan-out ℓ. The i-th input token is routed to output wire $i \bmod \ell$. Counting networks are constructed from balancers and from simple one-input one-output counting switches.

We denote by x_i, $0 \leq i \leq f_{in} - 1$, the number of tokens that have entered on input wire i, and similarly we denote by y_j, $0 \leq j \leq f_{out} - 1$, the number of tokens that have exited on output wire j. Figure 1(a) depicts a *binary* balancer. The balancer is represented by a "fat" vertical line.

On a shared memory multiprocessor, a switching network can be realized as a shared memory data structure, where switches are shared records and wires are pointers from one network switch to another. Each of the machine's n asynchronous processes runs a program that repeatedly "traverses" the data structure, each time shepherding a new token through the network. When a process reaches a switch, it atomically "updates" it, and uses the returned value to choose which pointer to follow. In a message-passing environment, each switch resides on a processor. Tokens are messages sent from processor to processor.

It is convenient to characterize a switch's internal state as a collection of variables, possibly with initial values. The state of a switch is given by its internal state and the collection of tokens on its input and output wires. Tokens also have state, also characterized by a set of variables. Tokens may change state as they traverse switches. It is convenient to assign each token a unique identifier (which may include the name of the process that owns that token). A switching network's state is just the collection of the states of its switches.

A switch is *quiescent* if the number of tokens that arrived on input wires equals the number that have exited on output wires: $\sum_{i=0}^{f_{in}-1} x_i = \sum_{j=0}^{f_{out}-1} y_j$. The *safety property* of a switch states that in any state, $\sum_{i=0}^{f_{in}-1} x_i \geq \sum_{j=0}^{f_{out}-1} y_j$; that is, a switch never creates tokens spontaneously. The *liveness property* states that given any finite number of input tokens to the switch, it is guaranteed that it will eventually reach a quiescent state; that is, a switch never "swallows" tokens. A switching network is *quiescent* if all its switches are quiescent.

We denote by $\pi = \langle t, b \rangle$ the *state transition* in which the token t passes from an input wire to an output wire of a switch b. Although transitions can occur concurrently, it is convenient to model them using an interleaving semantics in the style of Lynch and Tuttle [13]. If a token t is on one of the input wires of a switch b at some network state s, we say that t is *in front* of b at state s or that the transition $\langle t, b \rangle$ is *enabled* at state s. An *execution fragment* α of the network is either a finite sequence $s_0, \pi_1, s_1, \ldots, \pi_n, s_n$ or an infinite sequence $s_0, \pi_1, s_1, \ldots$ of alternating network states and transitions such that for each $\langle s_i, \pi_{i+1}, s_{i+1} \rangle$, the transition π_{i+1} is enabled at state s_i and carries the network to state s_{i+1}. If $\pi_{i+1} = \langle t, b \rangle$ we say that token t *takes a step* at state s_i (or that t *traverses* b at state s_i). An execution fragment beginning with an initial state is called an *execution*. If α is a finite execution fragment of the network and α' is any execution fragment that begins with the last state of α, then we write $\alpha \cdot \alpha'$ to represent the sequence obtained by concatenating α and α', eliminating the duplicate occurrence of the last state of α.

For any token t, a *t-solo execution fragment* is an execution fragment in all transitions of which token t only takes steps. A t-complete execution fragment is an execution fragment at the final state of which token t has exited the network. A finite execution is *complete* if it results in a quiescent state. An execution is *sequential* if for any two transitions $\pi = \langle t, b \rangle$ and $\pi = \langle t, b' \rangle$, all transitions between them also involve token t; that is, tokens traverse the network one completely after the other.

A switch b has the *c-balancing property* if, whenever c tokens reach each input wire of b then exactly c tokens exit on each of its output wires. We say that a switching network is a *c-balancing* network if all its switches preserve the c-balancing property. It can be proved [10] that in any execution α of a c-balancing network $\mathcal{N}$, in which no more than c tokens enter on any input wire of the network, there are never more than c tokens on any wire of the network.

For any execution α of a switching network $\mathcal{N}$, the *latency* of α is the maximum, over all tokens involved in α, of the number of switches traversed by the token in α. The *latency* of $\mathcal{N}$, denoted l, is the maximum, over all executions, of the latency of the execution. The *average* latency of a finite execution α is the sum, over all tokens involved in α, of the number of switches traversed by the token in α divided by the number of tokens involved in α. The *average* latency of an infinite execution α is the supremum over all non-empty prefixes of α of the number of switches traversed by any token involved in the prefix divided by the number of tokens involved in the prefix. The *average latency* of $\mathcal{N}$, denoted $\bar{l}$, is the maximum, over all executions, of the average latency of the execution.

An execution of a switching network $\mathcal{N}$ is called *one-shot* if each process shepherds its token through the network only once; that is, only n tokens, one for each process, are involved in the execution. For any execution α of a switching network $\mathcal{N}$, and for any switch $b \in \mathcal{N}$, the *contention of b* in α is the maximum, over all states of α, of the number of tokens that

are infront of b at the state. The *contention of* α is the maximum, over all switches, of the contention of the switch in α. The *one-shot contention of* $\mathcal{N}$, denoted c, is the maximum, over all one-shot executions of $\mathcal{N}$ for which the n tokens are uniformly distributed to the input wires, of the contention of the execution.

For any integer set S, an *S-adding network* is a switching network, each token t of which has been assigned an integer *weight* $\beta_t \in S$, and each execution of which satisfies the *adding property*. Let $l > 0$ be any integer and consider any complete execution α which involves l tokens $t_1, \ldots, t_l$. Assume that for each i, $1 \leq i \leq l$, β_i is the weight of t_i and v_i is the value taken by t_i in α. The *adding property* for α states that there exists a permutation $i_1, \ldots, i_l$ of $1, \ldots, l$, called *adding ordering*, such that (1) $v_{i_1} = 0$, and (2) for each j, $1 \leq j < l$, $v_{i_{j+1}} = v_{i_j} + \beta_{i_j}$; that is, the first token t_{i_1} returns the value zero, and each next token t_{i_j}, $1 < i_j \leq l$ returns the sum of the weights of its preceding tokens in the adding ordering. We say that a switching network is an *adding network* if it is a $\mathbb{Z}$-adding network, where $\mathbb{Z}$ is the set of integers.

3 Lower Bounds

Consider an (a, b)-adding network such that $|a| > |b| > 0$. We may assume without loss of generality that a and b have no common factors, since any (a, b)-adding network can be trivially transformed to an $(a \cdot k, b \cdot k)$-adding network, and vice-versa, for any non-zero integer k. Similarly, we can assume that a is positive.

We show that in any sequential execution, tokens of weight b must traverse at least $\lceil (n-1)/(c-1) \rceil$ switches, where c is the one-shot contention of the network. If $|b| > 1$, then in any sequential execution, tokens of weight a must also traverse the same number of switches.

Theorem 3.1 *Consider an (a, b)-adding network $\mathcal{A}$ where $|a| > |b| > 0$. Then, in any sequential execution of $\mathcal{A}$ (1) each token of weight b traverses $\Omega(\lceil (n-1)/(c-1) \rceil)$ switches, and (2) if $|b| > 1$ then each token of weight a also traverses $\Omega(\lceil (n-1)/(c-1) \rceil)$ switches.*

Proof: We prove something a little stronger, that the stated lower bound holds for any token that goes through the network by itself, independently of whether tokens before it have gone through sequentially.

Start with the network in a quiescent state s_0, denote by α_0 the execution with final state s_0, and let $t'_1, \ldots, t'_l$ be the tokens involved in α_0, where $l \geq 0$ is some integer. Denote by β_j, $1 \leq j \leq l$, the weight of token t'_j and let $v = \sum_{j=1}^{l} \beta_j$. The adding property implies that v is the next value to be taken by any token (serially) traversing the network. Let token t of weight x, $x \in \{a, b\}$, traverse the network next. Let $y \in \{a, b\}$, $y \neq x$, be the other value of $\{a, b\}$; that is, if $x = a$ then $y = b$, and vice versa. Because $|a| > |b| > 0$, $b \not\equiv 0 \pmod{a}$. Thus, if $x = b$, $x \not\equiv 0 \pmod{y}$. On the other hand, if $x = a$ it again holds that $x \not\equiv 0 \pmod{y}$ because by assumption $|y| > 1$ and x, y have no common factors.

Denote by $\mathcal{B}$ the set of switches that t traverses in a t-solo, t-complete execution fragment from s_0.

Consider $n-1$ tokens $t_1, \ldots t_{n-1}$, all of weight y. We construct an execution in which each token t_i, $1 \leq i \leq n-1$, must traverse some switch of $\mathcal{B}$.

Lemma 3.2 *For each i, $1 \leq i \leq n-1$, there exists a t_i-solo execution fragment with final state s_i starting from state s_{i-1} such that t_i is in front of a switch $b_i \in \mathcal{B}$ at state s_i.*

Proof: By induction on i, $1 \leq i \leq n-1$.

Basis Case

We claim that in the t_1-solo, t_1-complete execution fragment α_1' starting from state s_0, token t_1 traverses at least one switch of $\mathcal{B}$. Suppose not. Denote by s_1' the final state of α_1'. Because $\mathcal{A}$ is an adding network, t_1 takes the value v in α_1'.

Consider now the t-solo, t-complete execution fragment α_1'' starting from state s_1'. Since t_1 does not traverse any switch of $\mathcal{B}$, all switches traversed by t in α_1'' have the same state in s_0 and s_1'. Therefore, in α_1'' token t takes the same value as in the t-solo, t-complete execution fragment starting from s_0. It follows that t takes the value v.

We have constructed an execution in which both tokens t and t_1 take the value v. Since $|a|, |b| > 0$, this contradicts the adding property of $\mathcal{A}$.

It follows that t_1 traverses at least one switch of $\mathcal{B}$ in α_1'. Let α_1 be the shortest prefix of α_1' such that t_1 is in front of a switch $b_1 \in \mathcal{B}$ at the final state s_1 of α_1.

Induction Hypothesis

Assume inductively that for some i, $1 < i \leq n-1$, the claim holds for all j, $1 \leq j < i$: there exists an execution fragment with final state s_j starting from state s_{j-1} such that token t_j is in front of a switch $b_j \in \mathcal{B}$ at state s_j.

Induction Step

For the induction step, we prove that in the t_i-solo, t_i-complete execution fragment α_i' starting from state s_{i-1}, token t_i traverses at least one switch of $\mathcal{B}$. Suppose not. Denote by s_i' the final state of α_i'. Since t has taken no step in execution $\alpha_0 \cdot \alpha_1 \cdot \ldots \cdot \alpha_{i-1}$, the adding property of $\mathcal{A}$ implies that token t_i takes value $v_i \equiv v \pmod{y}$. Consider now the t-solo, t-complete execution fragment α_i'' starting from state s_i'. By construction of the execution $\alpha_0 \cdot \alpha_1 \cdot \ldots \cdot \alpha_{i-1}$, tokens $t_1, \ldots, t_{i-1}$ do not traverse any switch of $\mathcal{B}$ in $\alpha_0 \cdot \alpha_1 \cdot \ldots \cdot \alpha_{i-1}$. Therefore, all switches traversed by t in α_i'' have the same state at s_0 and s_i'. Thus, token t takes the value v in both α_i'' and in the t-solo, t-complete execution fragment starting from s_0.

Because $\mathcal{A}$ is an adding network, if t takes the value v, then t_i must take value $v_i \equiv v + x \pmod{y}$, but we have just constructed an execution where t takes value v, and t_i takes value $v_i \equiv v \pmod{y}$, which is a contradiction because $x \not\equiv 0 \pmod{y}$.

Thus, token t_i traverses at least one switch of $\mathcal{B}$ in α_i'. Let α_i be the shortest prefix of α_i' such that t_i is in front of a switch $b_i \in \mathcal{B}$ at the final state s_i of α_i, to complete the proof of the induction step.

At this point the proof of Lemma 3.2 is complete. ■

Let $\alpha = \alpha_0 \cdot \alpha_1 \cdot \ldots \cdot \alpha_{n-1}$. By Lemma 3.2, all tokens t_i, $1 \leq i \leq n-1$ are in front of switches of $\mathcal{B}$ at the final state of α. Notice also that all switches in $\mathcal{B}$ are in the same state at states s_0 and s_{n-1}. Thus, in the t-solo, t-complete execution fragment starting from state s_{n-1} token t traverses all switches of $\mathcal{B}$. Because $\mathcal{A}$ has contention c, no more than $c-1$ other tokens can be in front of any switch of $\mathcal{B}$ in α. Thus, $\mathcal{B}$ must contain at least $\lceil \frac{n-1}{c-1} \rceil$ switches. ■

Any S-adding network where S has at least three values is also an S'-adding network for any $S' \subseteq S$. Thus, Theorem 3.1 implies that in every sequential execution of the S-adding network all tokens traverse $\Omega(\lceil (n-1)/(c-1) \rceil)$ switches.

Corollary 3.3 *If $\mathcal{A}$ is an S-adding network for S containing at least three distinct non-zero values, then in any sequential execution of $\mathcal{A}$ all tokens traverse $\Omega(\lceil (n-1)/(c-1) \rceil)$ switches.*

We would like to point out that the one-shot contention of a large class of switching networks, including conventional counting networks, is constant. For example, consider the class of switching networks with $\Omega(n)$ input wires whose switches produce a permutation of their input tokens on their output wires. We can use a straightforward induction to prove that each switching network of this class has the 1-balancing property, and thus, in a one-shot execution, there are never more than one token on each of its wires. Therefore, the one-shot contention of such a network cannot be more than the maximum fan-in of any of its switches, and thus it is constant. By Theorem 3.1, all adding networks belonging to this class[†], present $\Omega(n)$ latency.

4 Upper Bounds

In this section, we show that the lower bound of Section 3 is essentially tight. We present a low-contention adding network, called LADDER, in which sequential executions traverse $O(n)$ switches, and concurrent executions traverse an average of $O(n)$ switches. For LADDER it holds that $c = 2$; that is, its contention is constant.

Topology

The switching network described here has the same topology as the SKEW counting network [10], though its behavior is substantially different.

A `Ladder` *layer* is an unbounded-depth switching network consisting of a sequence of binary switches b_i, $i \geq 0$. For switch b_0, both input wires are input wires to that layer, while for each switch b_i, $i > 0$, the north input wire is an output wire of switch b_{i-1}, while the south input wire is an input wire of the layer. The north output wire of any switch b_i, $i \geq 0$, is an output wire of the layer, while the south output wire of b_i is the north input wire of switch b_{i+1}.

A `Ladder` *switching network* with *layer depth* d is a switching network constructed by layering d `Ladder` layers so that the i-th output wire of the one is the i-th input wire to the next. Clearly, the `Ladder` switching network has an infinite number of input and output wires, numbered $0, \ldots, \infty$. The right hand-side of Figure 1 illustrates a `Ladder` switching network with layer depth 4. The LADDER *adding network* consists of a `Ladder` switching network *with layer depth* n, combined with a counting network. Each token first traverses the counting network, and then uses the resulting value as the index to its input wire into the infinite `Ladder` network. Thus, only one token enters on each input wire of the `Ladder` network.

[†]Most conventional counting networks belong to this class. Moreover, our LADDER network presented in Section 4, which is the first adding network studied thus far, is such a network.

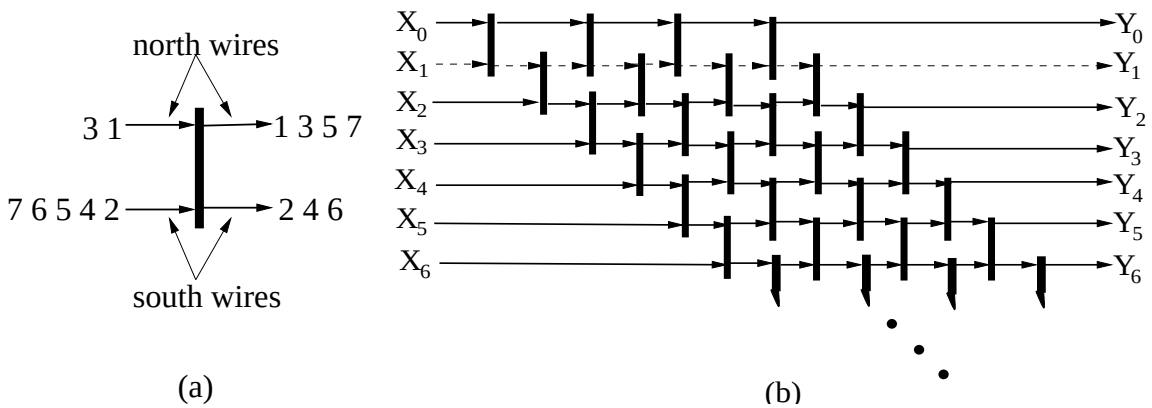

Figure 1: **(a)** A binary balancer. The balancer is represented by a "fat" vertical line. Its input and output wires are represented by horizontal arrows ending and starting at the balancer, respectively. The high (input/output) wires of a binary switch are also called the *north* wires, while its low wires are called the *south* wires. There are 7 tokens traversing the balancer. The tokens arrive on the balancer's input wires one after the other. For convenience, we have numbered them by the order of their arrival. **(b)** The `Ladder` switching network with layer depth 4. All switches for which one of their input wires is an input wire of the network belong to the first `Ladder` layer. The wires of *row* 1 are shown dashed.

We say that a wire of the `Ladder` network is on *row* k, $k > 0$, if it is the south input or output wire of the $(k-1)$st switch of any `Ladder` layer or if it is the kth output wire of the network. A wire is on row 0 if it is the north input or output wire of the first switch of any layer. We say that a token is on row k if it is on one of the wires of row k. In Figure 1(b), the wires of row 1 are shown dashed. (Note that higher-numbered rows appear lower on the page.)

States and Transitions

Each token begins by traversing a conventional counting network, and uses the result to choose an input wire to the `Ladder` network. The counting network ensures that each input wire is chosen by exactly one token, and each switch is visited by two tokens. A *fresh* switch is one that has never been visited by a token.

Within `Ladder`, a token proceeds in two *epochs*, a *North* epoch followed by a *South* epoch. Tokens behave differently in different epochs.

Each switch s has the following state: an $s.toggle$ bit that assumes values `north` and `south` (initially `north`), and an integer value $s.weight$, initially 0. The fields $s.north$ and $s.south$ are pointers to the (immutable) north and south output wires.

Each token t has the following state: The $t.arg$ field is the original weight of the token (argument to the initial counting network). The $t.weight$ field is originally 0, and it accumulates the sum of the weights of tokens ordered before t. The $t.wire$ field records whether the token will enter the next switch on its north or south input wire.

A token's North epoch starts when the token enters `Ladder`, continues as long as it traverses fresh switches, and ends as soon as it traverses a non-fresh switch. When a North-epoch token visits a fresh switch, the following occurs atomically (1) *s.toggle* flips from `north` to `south`, and (2) *s.weight* is set to *t.weight* + *t.arg*. Then, t exits on the switch's north wire.

The first time a token visits a non-fresh switch, it adds that switch's weight to its own, exits on the south wire, and enters its South epoch. If a token enters its South epoch on a particular row, then from then on it never moves "up" to a lower-numbered row. When a South-epoch token enters a switch on its south wire, it simply exits on the same wire (and same row), independently of the switch's current state. When a South-epoch token enters a switch on its north wire, it does the following. If the switch is fresh, then just as before, it atomically sets the token's weight to the sum of its weight and argument, flips the toggle bit, and exits on the North wire (same row). If the switch is not fresh, it adds the switch's weight to its own, and exits on the South wire (one row "down"). When the token emerges from LADDER, its current weight is its value.

Informally, the network works as follows. All tokens other than the one that exits on the first output wire reach a non-fresh switch. When a token t encounters its first non-fresh switch, then that switch's weight is the sum of all the tokens that will precede t (so far) in the adding ordering. Each time the token enters a non-fresh switch on its North wire, it has been "overtaken" by the earlier token, so it moves down one wire and adds the other's tokens weight to its own. Figure 2 shows pseudocode for the two epochs expressed in C notation. For ease of presentation, the pseudocode shows the switch complementing its *toggle* field and updating its *weight* field in one atomic operation. However, a slightly more complicated construction can realize this state change as a simple atomic complement operation on the toggle bit. Roughly speaking, this is implemented by having two weight variables associated with each switch, one for its north and one for its south input wire. A token first leaves its weight to the weight variable of the input wire on which it reaches the switch and then applies a `fetch&complement` operation on its toggle bit.

Even though the `Ladder` network has an unbounded number of switches, it can be implemented by a finite network by "folding" the network so that each folded switch simulates an unbounded number of primitive switches. A similar folding construction appears in [10].

LADDER is lock-free, but not wait-free. It is possible for a slow token to remain in the network forever if it is overtaken by infinitely many faster tokens.

For the LADDER network it holds that:

Proposition 4.1 *Consider a token t which exits* LADDER *on its kth output wire and let v be the value taken by t. Then, for each integer i, $0 \leq i \leq k-1$, (1) there exists exactly one token t_i that exits on output wire i of* LADDER*, (2) either t_i is in* LADDER *or it has exited* LADDER *by the time t exits the network, and (3) $v = \sum_{i=0}^{k-1} t_i.arg$.*

Sketch of proof: We first prove that the `Ladder` switching network is a 1-balancing network. This implies that at most one token ever traverses each of its wires. We then prove that all tokens other than the one which exits on the first output wire, reach a non-fresh switch at least once. This is so because `Ladder` has layer depth n, while additionally, by a well-known

```
void north_traverse(token t, switch s) {
    if (s.toggle == NORTH) { /* fresh */
        atomically {
            s.toggle = SOUTH;
            s.weight = t.weight + t.arg;
        }
        north_traverse(t, b.north);
    } else { /* not-so-fresh */
        t.weight += s.weight;
        t.wire = NORTH;
        south_traverse(t, b.south);
    }
}

void south_traverse(token t, switch s) {
    if (t.wire == SOUTH) { /* ignore switch */
        t.wire = NORTH; /* toggle wire */
        south_traverse(t, s.south);
    } else {
        t.wire = SOUTH; /* toggle wire */
        if (s.toggle == NORTH) { /* fresh */
            atomically {
                s.toggle = SOUTH;
                s.weight = t.weight + t.arg; /* leave my weight */
            }
            south_traverse(t, s.north);
        } else { /* overtaken */
            t.weight += s.weight; /* accumulate weight */
            south_traverse(t, s.south);
        }
    }
}
```

Figure 2: Pseudo-Code for LADDER Traversal

property [10, Lemma 2.1] of conventional counting networks it follows that when a token enters on some input wire of `Ladder`, there are at most $n-1$ lower-numbered wires on which no token has entered yet. Every time a token t visits a non-fresh switch b it moves south. If t then reaches row k, it will never reach any row less than k at the rest of the execution. Assuming that b belongs to some `Ladder` layer $\mathcal{L}$, we further prove that the path of t and the path of any token exiting on a lower output wire of $\mathcal{L}$ do not cross "at any later point". A token exiting on output wire 0 has never moved south and its *weight* is 0. All other tokens exit on the south

output wire of a switch at least once. We prove that every time a token t exits south on a switch of some layer reaching some row k, $t.weight$ is updated to the sum of the weights of all tokens that will exit on output wires less than k of the layer. We use these facts to conclude the stated claim. ■

Proposition 4.1 immediately implies that LADDER is an adding network. The adding ordering is derived by simply ordering tokens as they exited on network output wires.

Theorem 4.2 LADDER *is an adding network.*

The *traversal interval* of a token t is defined to be the time interval $[t_{enter}, t_{exit}]$ from the moment in which t entered the network until it exited it. A counting network is *linearizable* if for any two tokens t and t' which have traversed the network taking values v_t and $v_{t'}$ and have traversal intervals $[t_{enter}, t_{exit}]$ and $[t'_{enter}, t'_{exit}]$, respectively, it holds that if $t_{exit} < t'_{enter}$, then $v_t < v_{t'}$. For adding networks it holds that there exist executions which are linearizable (e.g., executions in which all tokens have different weights which are powers of two). For LADDER, Proposition 4.1(2) implies that any of its executions is linearizable.

Proposition 4.1(1) implies that only one token exits on each output wire. Thus, by having tokens reading the number of the output wire they exit, `Ladder` can also serve as a counting network (that is, by traversing `Ladder` just once, a token can obtain the result of both a `fetch&add` and a `fetch&inc` operations). It can be proved that `Ladder` can itself be used to play the role of the conventional counting network. From now on we assume that this is the case; that is, we assume that the LADDER adding network consists only of the `Ladder` switching network.

Proposition 4.1(1) implies that at the final state of any complete execution that involves l tokens, all tokens have exited on the l lower output wires. Since `Ladder` has layer depth n, each of the l wires traverses $2n$ switches. We can thus conclude the following theorem.

Theorem 4.3 *The average number of switches traversed by any token in* LADDER *is* $2n$.

Theorem 4.4 *In any sequential execution of* LADDER*, all tokens traverse exactly* $2n$ *switches.*

Sketch of proof: We prove by induction on k that in any sequential execution the kth token that exits the network enters on input wire k and never moves on any wire of a row other than k during its traversal. Since each row of LADDER traverses $2n$ switches, the theorem follows. ■

Recall that at most one token ever traverses each of the wires of LADDER. Thus, it follows that the contention of LADDER is constant.

Theorem 4.5 *The contention of* LADDER *is 2.*

5 Conclusion

We have introduced adding networks, a class of distributed data structures that support concurrent *fetch&add* operations. We have shown that adding networks require either high contention or high ($\Omega(n)$) latency, even if they add only two distinct values. This bound is tight: we present the first lock-free, low-contention adding network with average latency $O(n)$.

We close with some straightforward generalizations of our results. Recall that a *read-modify-write* operation [12] atomically replaces the variable value v with $\phi(v)$, for some function ϕ, and returns the prior value v. Our results are readily generalized to read-modify-write networks. Consider a family Φ of functions from values to values. Let ϕ be an element of Φ and x a variable. The *read-modify-write* operation, $RMW(x, \phi)$, atomically replaces the value of x with $\phi(x)$, and returns the prior value of x. Most common synchronization primitives, such as *fetch-and-add*, *swap*, *test-and-set*, and *compare-and-swap*, can be cast as read-modify-write operations for suitable choices of ϕ.

Define two maps ϕ and ψ to be *incompatible* if $\phi^k(\psi(x)) \neq \phi^\ell(x)$ for some value x and all natural numbers k and ℓ. Informally, one can always tell whether ψ has been applied to a variable, even after repeated successive applications of ϕ. For example, if ϕ is addition by a and ψ addition by b, where $|a| > |b| > 0$, then ϕ and ψ are incompatible.

Our lower bound can be generalized to show that if a switching network supports read-modify-write operations for only two incompatible functions, then in any n-process sequential execution, each process traverses $\Omega(n)$ switches before choosing a value.

On the other hand, the LADDER network is easily extended to a read-modify-write network for any family of *commutative* functions: for all ϕ, ψ in Φ, and all values v, $\phi(\psi(v)) = \psi(\phi(v))$.

Perhaps the most important remaining open question is whether there exist low-contention wait-free adding networks.

References

[1] E. Aharonson and H. Attiya, "Counting networks with arbitrary fan-out," *Distributed Computing,* Vol. 8, No. 4, pp. 163–169, 1995.

[2] W. Aiello, C. Busch, M. Herlihy, M. Mavronicolas, N. Shavit and D. Touitou, "Supporting Increment and Decrement Operations in Balancing Networks," *Proceedings of the 16th International Symposium on Theoretical Aspects of Computer Science,* pp. 393–403, Trier, Germany, March 1999.

[3] J. Aspnes, M. Herlihy and N. Shavit, "Counting Networks," *Journal of the ACM,* Vol. 41, No. 5, pp. 1020–1048, September 1994.

[4] C. Busch and M. Herlihy, "Sorting and Counting Networks of Small Depth and Arbitrary Width," *Proceedings of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures,* pp. 64–73, Saint-Malo, France, June 1999.

[5] C. Busch and M. Mavronicolas, "A Combinatorial Treatment of Balancing Networks," *Journal of the ACM,* Vol. 43, No. 5, pp. 794–839, September 1996.

[6] C. Busch and M. Mavronicolas, "An Efficient Counting Network," *Proceedings of the 1st Merged International Parallel Processing Symposium and IEEE Symposium on Parallel and Distributed Processing,* pp. 380–385, Orlando, Florida, May 1998.

[7] C. Dwork, M. Herlihy and O. Waarts, "Contention in Shared Memory Algorithms," *Journal of the ACM,* Vol. 44, No. 6, pp. 779–805, 1997.

[8] J. Goodman, M. Vernon and P. Woest, "Efficient synchronization primitives for large-scale cache-coherent multiprocessors," *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems,* pp. 64–75, Boston, Massachusetts, April 1989.

[9] M. Herlihy, B. Lim and N. Shavit, "Scalable Concurrent Counting," *Theory of Computing Systems Journal,* Vol. 13, No. 4, pp. 343–364, 1995.

[10] M. Herlihy, N. Shavit and O. Waarts, "Linearizable Counting Networks," *Distributed Computing,* Vol. 9, No. 4, pp. 193-203, 1996.

[11] M. Klugerman and C. Plaxton, "Small-Depth Counting Networks," *Proceedings of the 24th Annual ACM Symposium on Theory of Computing,* pp. 417–428, May 1992.

[12] C. Kruskal, L. Rudolph and M. Snir, "Efficient Synchronization on Multiprocessors with Shared Memory," *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing,* pp. 218–228, Calgary, Canada, August 1986.

[13] N. Lynch and M. Tuttle, "Hierarchical Correctness Proofs for Distributed Algorithms," *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing,* pp. 137–151, Vancouver, Canada, August 1987.

[14] M. Mavronicolas, M. Merritt and G. Taubenfeld, "Sequentially Consistent versus Linearizable Counting Networks," *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing,* pp. 133–142, May 1999.

[15] S. Moran and G. Taubenfeld, "A Lower Bound on Wait-Free Counting," *Journal of Algorithms,* Vol. 24, pp. 1–19, 1997.

[16] N. Shavit and D. Touitou, "Elimination trees and the Construction of Pools and Stacks," *Theory of Computing Systems,* Vol. 30, No. 6, pp. 645–670, 1997.

[17] N. Shavit and A. Zemach, "Diffracting Trees," *ACM Transactions on Computer Systems,* Vol. 14, No. 4, pp. 385–428, November 1996.