Adding Networks *

Panagiota Fatourou[1] and Maurice Herlihy[2]

[1] Department of Computer Science, University of Toronto, 10 King's College Road, Toronto, Canada M5S 3G4
faturu@cs.toronto.edu
[2] Department of Computer Science, Brown University, Box 1910, Providence, RI 02912-1910
mph@cs.brown.edu

Abstract. An *adding network* is a distributed data structure that supports a concurrent, lock-free, low-contention implementation of a *fetch&add* counter; a counting network is an instance of an adding network that supports only *fetch&increment*.
We present 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 a large class of switching networks and for conventional counting networks the one-shot contention is constant. On the contrary, counting networks have $O(\log n)$ latency [4, 7].
This bound is tight. We present the first concurrent, lock-free, low-contention networked data structure that supports arbitrary fetch&add operations.

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,

* This work has been accepted for publication as a brief announcement in the *20th Annual ACM Symposium on Principles of Distributed Computing,* Newport, Rhode Island, August 2001. Part of the work of the first author was performed while affiliating with the Max-Planck Institut für Informatik, Saarbrücken, Germany, and while visiting the Department of Computer Science, Brown University, Providence, USA.

Shavit and Touitou [11] 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? (A *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 nodes are called *switches*. Each of the n processes 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 any execution in which exactly n tokens enter the network on distinct wires. For counting networks, and for the (first) switching network presented in Section 4, 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). LADDER is the first concurrent, lock-free, low-contention networked data structure that supports arbitrary *fetch&add* operations.

An ideal *fetch&add* switching network (like an ideal counting network defined in [6]) 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:[1] a single switch is lock-free with low latency, but has high contention, a combining network [5] 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 by Aspnes *et. al* [3]. A flurry of research on counting networks followed (see e.g., [1, 2, 4, 6, 7, 9, 11]). 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 and tokens. 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 [6]. In [6] 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 [6], 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 introduced in Section 4.

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 b to switch b' is an *output wire* for b and an *input wire* for b'. A (w_{in}, w_{out})-switching network has w_{in} input wires and w_{out} output wires. A *switch* is a shared data object characterized by an internal state, its set

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

of f_{in} *input wires*, labeled $0, \ldots, f_{in}-1$, and its set of 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 move (shepherd) *tokens* through the network. Each process enters its token on one of the network's w_{in} input wires. After the token has traversed a sequence of switches, it leaves the network on one of its w_{out} output wires. 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. Processes work asynchronously, but they do not fail. In contrast to counting networks, associated to each token is a set of variables (that is, each 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 can 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. The wires are one-way communication channels and allow reordering. Communication is asynchronous but reliable (meaning a token does not wait on a wire forever). For each (f_{in}, f_{out})-switch, 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.

As an 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.

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. Each token's state is also characterized by a set of variables. Notice that a token's state is part of the state of the process owning it. A process may change the state of its token while moving it through a switch. 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 its input wires equals the number that have exited on its 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. 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 is moved from an input wire to an output wire of a switch b. 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' and 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 *l-balancing property* if, whenever l tokens reach each input wire of b then exactly l tokens exit on each of its output wires. We say that a switching network is an *l-balancing* network if all its switches preserve the l-balancing property. It can be proved [6] that in any execution α of an l-balancing network $\mathcal{N}$, in which no more than l tokens enter on any input wire of the network, there are never more than l tokens on any wire of the network.

The *latency* of a switching network is the maximum number of switches traversed by any single token in any execution. The *contention of an execution* is the maximum number of tokens that are on the input wires of any particular switch at any point during the execution. The *contention* of a switching network is the maximum contention of any of its executions. In a *one-shot* execution, only n tokens (one per process) traverse the network. The *one-shot contention* of a switching network, denoted c, is the maximum contention over all its one-shot executions in which the n tokens are uniformly distributed on the input wires. For counting networks with $\Omega(n)$ input wires, and for the switching network presented in Section 4, c is constant.

For any integer set S, an *S-adding network* $\mathcal{A}$ is a switching network that supports the operation *fetch&add*$(\cdot, v)$ only for values $v \in S$. More formally, 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 the *adding order*, 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 in the order returns the value zero, and each subsequent token returns the sum of the weights of the tokens that precede it. 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 Bound

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 (involving any number of tokens), 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.

We remark that our lower bound holds for all (a, b)-adding networks, independently of e.g., the topology of the network (the width or the depth of the network, etc.) and the state of both the switches and the tokens. Moreover, our lower bound holds for *all* sequential executions involving any number of tokens (and not only for one-shot executions).

Theorem 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 at least $\lceil (n-1)/(c-1) \rceil$ switches, and*
2. *if $|b| > 1$ then each token of weight a traverses at least $\lceil (n-1)/(c-1) \rceil$ switches.*

Proof. We prove something slightly stronger, that the stated lower bound holds for any token that goes through the network alone, independently of whether tokens before it have gone through the network 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 $\mathcal{B}$. Assume that all n tokens $t, t_1, \ldots, t_{n-1}$ are uniformly distributed on the input wires.

Lemma 1. *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, token t takes the same value in α_1'' 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, $1 \leq j < i$; that is, there exists an execution fragment a_j 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 $\alpha_1 \cdot \ldots \cdot \alpha_{i-1}$, 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 $\alpha_1 \cdot \ldots \cdot \alpha_{i-1}$, tokens $t_1, \ldots, t_{i-1}$ do not traverse any switch of $\mathcal{B}$ in $\alpha_1 \cdot \ldots \cdot \alpha_{i-1}$. 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 1 is complete.

Let $\alpha = \alpha_1 \cdot \ldots \cdot \alpha_{n-1}$. Clearly, only n tokens, one per process, are involved in α and they are uniformly distributed on the input wires, so α is a one-shot execution. By Lemma 1, 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 one-shot 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.

Since any S-adding network, where $|S| > 2$, is an S'-adding network for all $S' \subseteq S$, Theorem 1 implies that in every sequential execution of the S-adding network all tokens (except possibly those with maximum weight) traverse $\Omega(n/c)$ switches.

We remark that the one-shot contention c 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 any permutation of their input tokens on their output wires. A straightforward induction argument shows that each switching network of this class has the 1-balancing property, and thus in a one-shot execution it never has more than one token on each wire. It follows that the one-shot contention of such a network is bounded by the maximum fan-in of any of its switches. By Theorem 1, all adding networks in this class, have $\Omega(n)$ latency. Conventional counting networks (and the LADDER adding network introduced in Section 4) belong to this class.

4 Upper Bound

In this section, we show that the lower bound of Section 3 is essentially tight. We present a low-contention adding network, called LADDER, such that in any of its sequential executions tokens traverse $O(n)$ switches, while in its concurrent executions they traverse an *average* of $O(n)$ switches. The switching network described here has the same topology as the SKEW counting network [6], though its behavior is substantially different.

A `Ladder` *layer* is an unbounded-depth switching network consisting of a sequence of *binary* switches b_i, $i \geq 0$, that is, switches with $f_{in} = f_{out} = 2$. For switch b_0, both input wires are input wires to the layer, while for each switch b_i, $i > 0$, the first (or *north*) input wire is an output wire of switch b_{i-1}, while the second (or *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* of *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. The LADDER *adding network* consists of a counting network followed by a `Ladder` switching network *of layer depth* n.

Figure 1 illustrates a `Ladder` switching network of layer depth 4. Switches are represented by fat vertical lines, while wires by horizontal arrows. Wires $X_0, X_1, \ldots$, are the input wires of the network, while $Y_0, Y_1, \ldots$, are its output wires. All switches for which one of their input wires is an input wire of the network belong to the first `Ladder` layer. All dashed wires belong to *row* 1.

Each process moves its token through the counting network first, 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. Each switch s has the following state: a bit *s.toggle* 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) switches that are connected to s through its north and south output wires, respectively.

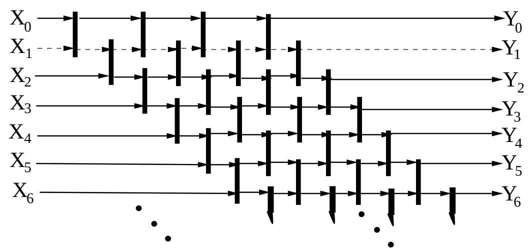

Fig. 1. The `Ladder` switching network of layer depth 4.

Each token t has the following state: the $t.arg$ field is the original weight of the token. 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.

Within `Ladder`, a token proceeds in two *epochs*, a *north* epoch followed by a *south* epoch. Tokens behave differently in different epochs. 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. Once a token enters its south epoch, 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, as before, it atomically sets the switch'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 output value.

All tokens other than the one that exits on the first output wire eventually 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 order. Each time the token enters a non-fresh switch on its north wire, it has been "overtaken" by an earlier token, so it moves down one row and adds this other token's weight to its own. Figure 2 shows pseudo-code for the two epochs. 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.

Even though `Ladder` 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 [6].

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.

```
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;
            }
            south_traverse(t, s.north);
        } else { /* overtaken */
            t.weight += s.weight;
            south_traverse(t, s.south);
        }
    }
}
```

Fig. 2. Pseudo-Code for LADDER Traversal.

Proving that LADDER is an adding network is a major challenge of our analysis. We point out that although LADDER has the same topology as SKEW [6], LADDER is substantially more powerful than SKEW; we require a substantially different and more complicated analysis to prove its adding property.

Theorem 2. LADDER *is an adding network.*

The performance analysis of LADDER uses similar arguments as the one of SKEW. This follows naturally from the fact that the two networks have the same topology and they both maintain the 1-balancing property (notice that, by knowing just the topology of a switching network, it is not always possible to analyze its performance because the way each token moves in the network may depend on both the state of the token and the state of any switch it traverses).

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, the LADDER adding network consists only of the `Ladder` switching network (which it also uses as a traditional counting network).

Theorem 3. (a) *In any execution of* LADDER*, each token traverses an average number of* $2n$ *switches;*
(b) *In any sequential execution of* LADDER*, each token traverses exactly* $2n$ *switches;*
(c) *The contention of* LADDER *is 2.*

An execution of a switching network is *linearizable* if for any two tokens t and t' such that t' entered the network after t has exited it, it holds that $v_t < v_{t'}$, where $v_t, v_{t'}$ are the output values of t and t', respectively. For any adding network *there exist* executions which are linearizable (e.g., executions in which all tokens have different weights which are powers of two). For LADDER it holds that *any* of its executions is linearizable. It has been proved [6, Theorem 5.1, Section 5] that any non-blocking linearizable counting network other than the trivial network with only one balancer has infinite number of input wires; that is, if *all* the executions of the network are linearizable, then the network has infinite width. Although an adding network implements a *fetch&increment* operation (and thus it can serve as a counting network), this lower bound does not apply for adding networks because its proof uses the fact that counting networks consist only of balancers and counter objects which is not generally the case for switching networks.

5 Discussion

We close with some straightforward generalizations of our results. 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 [8], $RMW(x, \phi)$, atomically replaces the value of x with $\phi(x)$, and returns the prior value of x. Most common synchronization primitives, such as *fetch&add*, *swap*, *test&set*, and *compare&swap*, can be cast as *read-modify-write* operations for suitable choices of ϕ. A *read-modify-write network* is one that supports *read-modify-write* operations. The LADDER network is easily extended to a read-modify-write network for any family of *commutative* functions (for all functions $\phi, \psi \in \Phi$, and all values v, ϕ and ψ are *commutative*, if and only if $\phi(\psi(v)) = \psi(\phi(v))$).

A map ϕ can *discern* another map ψ 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 can discern ψ. Our lower bound can be generalized to show that if a switching network supports read-modify-write operations for two functions one of which can discern the other, then in any n-process sequential execution, processes traverse $\Omega(n/c)$ switches before choosing a value.

Perhaps the most important remaining open question is whether there exist low-contention wait-free adding networks (notice that since switching networks do not contain cycles, any network with a finite number of switches would be wait-free).

Acknowledgement: We would like to thank Faith Fich for many useful comments that improved the presentation of the paper. Our thanks go to the anonymous DISC'01 reviewers for their feedback.

References

1. Aharonson, E., Attiya, H.: Counting networks with arbitrary fan-out. *Distributed Computing,* **8** (1995) 163–169.
2. Aiello, W., Busch, C., Herlihy, M., Mavronicolas, M., Shavit, N., Touitou, D.: 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. Aspnes, J., Herlihy, M., Shavit, N.: Counting Networks. *Journal of the ACM,* **41** (1994) 1020–1048.
4. Busch, C., Mavronicolas, M.: 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.
5. Goodman, J., Vernon, M., Woest, P.: 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.
6. Herlihy, M., Shavit, N., Waarts, O.: Linearizable Counting Networks. *Distributed Computing,* **9** (1996) 193–203.
7. Klugerman, M., Plaxton, C.: Small-Depth Counting Networks. *Proceedings of the 24th Annual ACM Symposium on Theory of Computing,* pp. 417–428, May 1992.
8. Kruskal, C., Rudolph, L., Snir, M.: 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.
9. Mavronicolas, M., Merritt, M., Taubenfeld, G.: Sequentially Consistent versus Linearizable Counting Networks. *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing,* pp. 133–142, May 1999.
10. Moran, S., Taubenfeld, G.: A Lower Bound on Wait-Free Counting. *Journal of Algorithms,* **24** (1997) 1–19.
11. Shavit, N., Touitou, D.: Elimination trees and the Construction of Pools and Stacks. *Theory of Computing Systems,* **30** (1997) 645–670.
12. Wattenhofer, R., Widmayer, P.: An Inherent Bottleneck in Distributed Counting. *Journal of Parallel and Distributed Computing,* **49** (1998) 135–145.