Alcom-FT Technical Report Series
ALCOMFT-TR-01-76

Brief Announcement: Adding Networks*

Panagiota Fatourou
Department of Computer Science
University of Toronto
faturu@cs.toronto.edu

Maurice Herlihy
Department of Computer Science
Brown University
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. We give a lower bound showing that adding networks have inherently high *latency*. We prove that our lower bound is tight.

1. INTRODUCTION

A *counting network* [2] is a class of distributed data structures used to construct concurrent, low-contention implementations of *fetch&increment* counters. Modern counting networks support both increment and decrement operations [1]. In this paper, we consider the problem of whether we can construct network data structures that support lock-free, highly-concurrent, low-contention *fetch&add* operations. We address this problem in the context of concurrent *switching networks*, a generalization of the balancing networks used to construct counting networks.

A *switching network* 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'. There are n processes that move *tokens* through the network; each *token* enters on one of the network's input wires, traverses a sequence of switches, and leaves on one of the network's output wires. Each process has an arbitrary number of distinct tokens but it cannot move more than one through the network concurrently. A process can maintain a set of variables for each of its tokens which comprise the *token's state*. The state of a token can change as it is moved through the network. A *switch* is a shared data object characterized by an internal state, and its set of f_{in} *input* wires and f_{out} *output wires*. (The values f_{in} and f_{out} are called the switch's *fan-in* and *fan-out* respectively). 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. Processes work asynchronously but they do not fail.

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, each token t of $\mathcal{A}$ has been assigned an integer *weight* $\beta_t \in S$, and each of its execution satisfies the following property: there exists a permutation of the tokens of the execution, called the *adding order*, such that (1) 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. A switching network is an *adding network* if it is a $\mathbb{Z}$-adding network, where $\mathbb{Z}$ is the set of integers.

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 3, c is constant.

An ideal *fetch&add* switching network is (1) lock-free, with (2) low contention, and (3) low latency. In Section 2, we prove that no switching network can have all three properties. However, any two are possible: a trivial switching network consisting of a single switch is lock-free with low latency but has high contention; a combining network [3] has low contention and $O(\log n)$ latency but requires tokens to wait for one another, and the construction presented in Section 3 is lock-free with low contention but has $O(n)$ latency. (NASA's motto "faster, cheaper, better" has been satirized as "faster, cheaper, better: pick any two".)

*The work of the first author was done while affiliating with the Max-Planck Institut für Informatik, Saarbrücken, Germany, and while visiting the Department of Computer Science of Brown University, Providence, and was partially supported by the IST Programme of the European Union under contract number IST-1999-14186 (ALCOM-FT).

More specifically, in Section 2 we show that for any S-adding network, there exist sequential executions in which each process traverses $\Omega(n/c)$ switches. Thus, any lock-free, low-contention adding network must have high latency, even in the absence of concurrency. However, our results encompass not only bad but also good news. In Section 3, we introduce LADDER, the first concurrent, lock-free, low-contention adding network. We prove that, in LADDER, processes take $O(n)$ steps on average.

2. LOWER BOUND

Consider an $\{a, b\}$-adding network $\mathcal{A}$, where $|a| > |b| > 0$. 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.

THEOREM 1. *In any sequential execution of $\mathcal{A}$, (a) each token of weight b traverses at least $\lceil (n-1)/(c-1) \rceil$ switches, and (b) if $|b| > 1$ then each token of weight a also traverses at least $\lceil (n-1)/(c-1) \rceil$ 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 one specific weight value) traverse $\Omega(n/c)$ switches.

We remark 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. It can be easily proved that the one-shot contention of such a network is bounded by the maximum fan-in of any of its switches. Thus, by Theorem 1, all adding networks in this class have $\Omega(n)$ latency.

3. UPPER BOUND

In this section, we show that the lower bound of Section 2 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 [4], 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 that 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

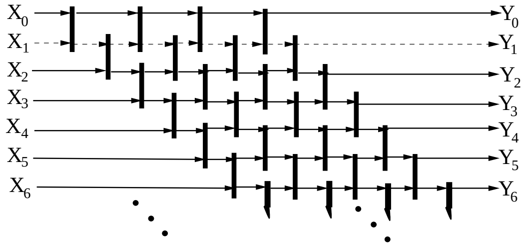

Figure 1: The `Ladder` switching network with layer depth 4.

number of input and output wires. The LADDER *adding network* consists of a counting network followed by a `Ladder` switching network *with layer depth n*.

Figure 1 illustrates a `Ladder` switching network with layer depth 4. Switches are represented by fat vertical lines, while wires by horizontal arrows. The high (low) wires of a switch are its *north* (respectively, *south*) wires. 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.

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,

```
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);
        }
    }
}
```

Figure 2: Pseudo-Code for LADDER Traversal.

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 ordering. 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.

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 [4]. 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.

THEOREM 2. *(a)* LADDER *is an adding network, (b) in any execution of* LADDER*, each token traverses an average number of $2n$ switches, (c) in any sequential execution of* LADDER*, each token traverses exactly $2n$ switches, and (d) the contention of* LADDER *is 2.*

A counting 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 [4, 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.

4. 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, $RMW(x, \phi)$, 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.

5. REFERENCES

[1] W. Aiello, C. Busch, M. Herlihy, M. Mavronicolas, N. Shavit, and D. Touitou. Supporting increment and decrement operations in balancing networks. In *Proceedings of the 16th International Symposium on Theoretical Aspects of Computer Science*, pages 393–403, Trier, Germany, March 1999.

[2] J. Aspnes, M. Herlihy, and N. Shavit. Counting networks. *Journal of the ACM*, 41(5):1020–1048, 1994.

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

[4] M. Herlihy, N. Shavit, and O. Waarts. Linearizable counting networks. *Distributed Computing*, 9(4):193–203, 1996.