A lower bound on wait-free counting

S. Moran, G. Taubenfeld

Computer Science/Department of Algorithmics and Architecture

Report CS-R9307 February 1993

CWI is the National Research Institute for Mathematics and Computer Science. CWI is part of the Stichting Mathematisch Centrum (SMC), the Dutch foundation for promotion of mathematics and computer science and their applications. SMC is sponsored by the Netherlands Organization for Scientific Research (NWO). CWI is a member of ERCIM, the European Research Consortium for Informatics and Mathematics.

P.O. Box 4079, 1009 AB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

A Lower Bound on Wait-Free Counting

Shlomo Moran
CWI
P.O. Box 4079, 1009 AB
Amsterdam, The Netherlands
and
Computer Science Department
Technion, Haifa 32000, Israel.

Gadi Taubenfeld
AT&T Bell Laboratories
600 Mountain Avenue, Murray Hill, NJ 07974, USA

Abstract

A counting protocol (mod m) consists of shared memory bits - referred to as the counter - and of a procedure for incrementing the counter value by 1 (mod m). The procedure may be executed by many processes concurrently. It is required to satisfy a very weak correctness requirement, namely: the counter is required to show a correct value only in quiescent states – states in which no process is incrementing the counter. Special cases of counting protocols are "counting networks" [AHS91] and "concurrent counters" [MTY92].

We consider the problem of implementing a wait-free counting protocol, assuming that the basic atomic operation of a process is a read-modify-write on a single bit. Let $flip(Pr)$ be the maximum number of times a single increment operation changes the counter bits in a counting protocol Pr. Our main result is: In any wait-free counting protocol Pr which counts modulo m, m divides $2^{flip(Pr)}$. Thus, $flip(Pr) \geq \log m$ and m is a power of 2.

This result provides interesting generalizations of lower bounds and impossibility results for counting and smoothing networks.

1991 Mathematics Subject Classification: 68P99,68Q22,68Q25.
Keywords & Phrases: Wait Free Protocols, Counting Protocols, Counting and Smoothing Networks, Lower Bounds.
*Note:*Part of this work was done while the first author was visiting at AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, USA.

1 Introduction

Recently there was much interest in the implementation of counters in a concurrent environment where many processes may try to access the counter, possibly at the same time. The goal is to come up with a solution that reduces memory contention and achieves high level of concurrency. The interest in the subject arise from the fact that such counters can be used to efficiently solve various coordination problems.

In this paper we prove a basic lower bound on implementing such counters, for systems which consist of a fully asynchronous collection of identical processes that communicate

Report CS-R9307
ISSN 0169-118X
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

via shared registers. Access to a shared register is via an atomic "read-modify-write" instruction, which, in a single indivisible step, reads the value of a single bit and then writes a new value that can depend on the value just read.

Our result applies to any implementation that supports an operation which enables to increment a counter by one in a wait-free manner. That is, an increment operation initiated by a correct process must terminate, regardless of the speed of all the other processes.

We require a counter to satisfy a very weak correctness requirement, namely: the counter is required to show a correct value only in *quiescent* states – states in which no process is incrementing the counter. Notice that the increment operation is not required to return a value. Both "counting networks" [AHS91] and "concurrent counters" [MTY92] satisfy this correctness requirement.

In fact, counting networks and concurrent counters satisfy much stronger correctness requirements. In counting networks the increment operation also returns the current value of the counter, and such networks also support an operation which returns a correct value when the system is in a quiescent state. Concurrent counters (which count modulo m) support an independent look operation which returns a value such that: if r processes have started an increment operation, and ℓ of them have completed their increment operations, then the value return by the look operation is in the cyclic interval $[\ell, r] \bmod m$.

We use the notion *counting protocol* for an implementation of a counter which satisfies the weak correctness requirement mentioned earlier, and which enables to count modulo some fixed number m. (In counting networks m is the number of output wires.) Let $flip(Pr)$ be the maximum number of times a single increment operation changes the counter bits in a counting protocol Pr; and let $space(Pr)$ is the number of shared registers used for the counter. The main result reported here can now be stated as follows: [1]

> *Let Pr be a wait free counting protocol which counts modulo m, and assume that both $space(Pr)$ and $flip(Pr)$ are finite. Then m divides $2^{flip(Pr)}$. Thus, $flip(Pr) \geq log(m)$ and m is a power of 2.*

Let $depth(Pr)$ be the maximum number of times a single increment operation accesses the counter bits in the counter Pr. Since it is possible to access a bit without changing its value, $flip(Pr) \leq depth(Pr)$. Hence a lower bound on $flip(Pr)$ implies a similar lower bound on $depth(Pr)$, but not vice versa.

One application of our result is that if Pr is a counting protocol mod m where m is not a power of 2 and Pr uses bounded shared space, then $flip(Pr)$ is not finite. This result generalizes a similar result for acyclic counting networks ([AA92]), since both acyclic and cyclic counting networks are (special cases of) counting protocols.

Using the observation that if a balancing network counts, then its isomorphic comparison network sorts [AHS91, Theorem 2.4], and the fact that the depth of any *sorting* network is at least $\log m$, one can immediately derive that the depth of any *counting* network is at least $\log m$, where m is the number of output (and input) wires. We notice that the $\log m$ lower bound on the depth of counting networks follows directly from our result. To see why, we observe that any counting network, Pr, is also a counting protocol, where $flip(Pr) = depth(Pr)$. Hence we get that $depth(Pr) = flip(Pr) \geq \log m$.

The above argument is not applicable to K-smoothing networks, which are a generalized form of a counting network. (The definition is given in Section 5.) Every counting network is a K-smoothing network for all $K \geq 1$, but the converse is not necessarily true. In particular, it is not necessarily the case that K-smoothing networks are isomorphic to sorting networks. We observe that a K-smoothing network can be transformed into a counting protocol (not a counting network) by adding $\lceil \log(2K+1) \rceil$ bits at each output wire. We define a class

[1] All logarithms are to base two.

of *K-smoothing protocols*, which generalizes the notion of K-smoothing networks, and we use our result to show that K-smoothing protocols are possible only when m is a power of 2, and they require at least $\log m - \lceil \log(2K+1) \rceil$ bit changes per token distribution. We will also provide a simple direct proof that the depth of K-smoothing networks is at least $\log m$.

The above results indicate that relatively tight lower bounds on counting and smoothing networks can be obtained by using only a part of the correctness requirements they are required to satisfy. It should be noted that counting networks are a severely restricted form of counting protocols - this is vividly demonstrated by the fact that a counting protocol can be implemented using only $\log m$ bits, while a counting network require $\Omega(m \log m)$ bits.

Our result implies that for every counting protocol Pr, $flip(Pr) \geq \log m$. An immediate question is whether this bound id tight. The answer is positive, since in the Positional counter presented in [MTY92], $depth(Pr) = flip(Pr) = \log m$, and it works for any number of concurrent increment operations using $\log m$ bits.

Finally, we point out that the result stated in Lemma 4.1 is stronger than our result as stated above, since it relates the number of processes which can increment the counter and the total number of bits used, and hence is applicable also to protocols in which only a bounded number of processes may increment the counter.

1.1 Related Work

Aspnes, Herlihy and Shavit [AHS91] introduced a new class of networks, called *counting networks*. Counting networks can be viewed as objects which support one atomic operation, which consists of both incrementing and reading the value of a counter. The two constructions of counting networks in [AHS91] require $O(m \log^2 m)$ binary registers. Counting networks achieve a high level of throughput by decomposing interactions among processes into pieces that can be performed in parallel, effectively reducing memory contention. Experimental results of implementing shared counters, producer/consumer buffers and barrier synchronization, show that counting network implementations provide higher throughput, less memory contention, and better fault-tolerance, compared to conventional implementations based on spin locks [AHS91]. Smoothing networks were defined in the context of counting networks, and may be used as building blocks for counting networks [AHS91].

Counting networks have been the subject of intensive investigation recently [AA92, AHS91, HBS92, HSW91, KP92]. A tight bound of $\Omega(m \log m)$, on the number of 2-balancers needed to construct counting networks is proved in in [KP92]. In [AA92] it is proven, among other results, that using 2-balancers it is impossible to construct *acyclic* counting networks with fan-out which is not a power of 2. The authors overcome this limitation for *acyclic* networks by presenting a construction of *cyclic* counting networks with fan out n, for arbitrary n. Unfortunately, their construction is not wait free – increment operations are not guaranteed to ever terminate. Our result implies that even in the implementation of an object which is considerably weaker than counting network - namely, a counting protocol - an increment operation cannot be terminated within a fixed number of bit changes, unless it counts modulus m where m is a power of 2.

In [HSW91] two interesting lower bounds are introduced for linearizable counting. Linearizable counting impose an additional restriction: the order in which values are assigned by the protocol reflects the real time order in which they were requested.[2] These results indicates that there exists a substantial complexity gap between linearizable and non-linearizable counting.

Independently of the work on counting networks, the notion of concurrent counters was introduced in [MTY92]. Concurrent counter (mod m) holds an integer from $0, .., m-1$, and

[2] As in counting networks, the paper assumes an increment operation that also returns the value of the counter, and the "value assigned" refers to the value returned by the increment operation.

enables two operations: `increment` – which increments the value by one (mod m), and `look` – which gets the current value. Two types of counters are considered: (1) *static counters* which only guarantee that a `look` operation returns the correct value only at quiescent states, and (2) *dynamic counters* which also guarantee that processes can read a correct value of the counter even if the read is concurrent with some increments.

Our result here implies that when the number of incrementors is unbounded, static (and hence also dynamic) counters (mod m) are possible only when m is a power of two, and that an increment operation in such counters requires at least $\log m$ bit flips. This should be contrasted with the fact that when the number of incrementors is bounded, it is possible to construct *dynamic* counting protocols, in which each increment operation requires essentially one bit change (variants of "1-flip protocols" [MTY92]).

2 Definitions and Notations

In a first reading of the paper, the reader may wish to skip this section and proceed immediately to the results in later sections.

A *counting protocol* consists of a collection of shared binary registers $r_1, ..., r_k$, which we refer to as a *counter*, a function *val* that associates some integer value – the value of the counter – in the range $\{0, ..., m-1\}$ to any possible contents of the counter, and a procedure, called increment , for incrementing the counter. It is also assumed that a specific vector is an *initial contents* of the counter.

Formally, a counting protocol (mod m) over k bits is a triple (increment, val, $\vec{v}_{init}$), where:

1. increment is a procedure for incrementing the counter;
2. $val : \{0,1\}^k \rightarrow \{0, \cdots, m-1\}$ is a function, which assigns to each binary k-vector a value in $\{0, \cdots, m-1\}$;
3. $\vec{v}_{init} = (v_1, \cdots, v_k)$ is the initial contents of the counter, and $val(\vec{v}_{init}) = 0$.

A process performs the increment operation on the counter by executing an increment procedure. (Many increments can take place concurrently.) Each increment operation consists of a sequence of atomic read-modify-write (in short *rmw*) steps applied to the counter registers. In a single *rmw* step a process may reads the value of a single bit and then writes a new value that can depend on the value just read.

A *run* starts from some initial values of the counter registers and consists of a finite or infinite sequence of atomic *rmw* steps. A run is *legal* if it starts when the contents of the counter is $\vec{v}_{init}$. A process p is *involved* in an increment operation in a given run if it has started an increment operation but have not completed it yet. A process is involved in an increment operation *during* a run, if it is involved in that operation in some prefix of that run. A run is *complete* if no process is involve in an increment operation in it.

A wait-free counting protocol (increment, val, $\vec{v}_{init}$) satisfies the following two requirements:

correctness Let $final(x)$ be the contents of the counter registers at the end of a run x. Then, for every complete legal run x, $val(final(x))$ equals to the number of increment operations terminated in the run x (modulo m).

wait-freedom In every legal run in which a process takes infinitely many steps, all its increment operations are completed. That is, a process that starts an increment operation eventually completes it, regardless of the activity of the other processes.

We study two complexity measures related to counting protocol Pr: $space(Pr)$ is the number of shared registers, k, used for the counter, and $flip(Pr)$ is the supremum on the

number of times a bit can be flipped by a single increment operation, taken over all the legal runs of Pr. Note that $flip(Pr)$ may be infinite. The notion $flip(Pr)$ is related to the notion $depth(Pr)$, which is the supremum on the number of times a single increment operation accesses the counter bits. Since it is possible to access a register without changing its value, $flip(Pr) \leq depth(Pr)$, where strict inequality is possible. Thus, any lower bound on $flip(Pr)$ implies a similar lower bound on $depth(Pr)$, but not vice-versa.

3 The Hiding Lemma

In this section we show how up to $2^{flip(Pr)}$ increment operations, each of which executed by a different process, can be hidden in some run of a counter protocol Pr. The operations are hidden in the sense that once they all terminate they have no effect on the counter registers, and all the processes not involved in these increment operations have no way to know that these $2^{flip(Pr)}$ increment operations actually took place.

Definition 1 *A group of processes G is* hidden *in run x if the subsequence x' of all events in x involving processes not in G is a run, and the value of the counter in x and in x' is the same.*

In the definition of counting protocol no bound is assumed on the number of processes that may increment the counter, that is, the number of processes that may increment the counter is assumed to be infinite. In order to state our results in the strongest possible way, we use the notion of an n-counting protocol, which is a protocol that behaves as a correct counting protocol as long as the number of processes that increment the counter is no more then n (each process may increment the counter many times). More formally, an n-counting protocol must satisfy the wait freedom requirement, and for any complete legal run x in which at most n processes participate, the value of the counter at x equals to the number of processes that increment the counter in x (mod m). Recall that for a counting protocol Pr, we denote by $space(Pr)$ the size of the counter (number of bits) used by Pr.

Lemma 3.1 *(The Hiding Lemma) Let Pr be a wait-free n-counting protocol, let $k = space(Pr)$ and $f = flip(Pr)$. If $k \leq \frac{n-2^f}{2^f-1}$, then there exists a complete run, ρ, in which a group of 2^f processes is hidden, and each process in this group has completed one increment operation, and is idle in ρ.*

Proof: Assume that $k \leq \frac{n-2^f}{2^f-1}$. We construct a complete run ρ which satisfies the lemma. For $0 \leq i \leq f$, let $n_i = k \cdot (2^{f-i} - 1) + 2^{f-i}$. Notice that $n_0 \leq n$, $n_f = 1$ and $n_{i+1} = \frac{n_i - k}{2}$. Thus, for $0 \leq i < f$ it holds that $n_i > k$.

We build a sequence of runs $\rho_0, \cdots, \rho_f$, where $\rho = \rho_f$. The construction is carried by induction, in rounds. In each round $0 < i \leq f$ we extend the run ρ_{i-1} built in the previous round to a run ρ_i so that: there is a group G_i of processes that is *hidden* in ρ_i, the size of G_i is $2^i n_i$, each process in G_i has already flipped the counter registers i times or has completed its increment operation in ρ_i, and all processes not in G_i are idle. The run ρ_f is ρ. Notice that since $f = flip(Pr)$, we get that all processes are idle in ρ_f, and a group of $|G_f| = 2^f$ processes is *hidden* in ρ_f, as required.

Round 0: In Round 0, the run ρ_0 is the empty run, and G_0 includes n_0 processes. ($n_0 \leq n$.) Next we show how ρ_1 is built.

Round 1: Consider a run α where $k+1$ different processes are activated in a sequential manner, one at a time starting from some arbitrary initial state. Each of these $k+1$ processes when activated starts an increment operation and is delayed once it changes one of the counter registers for the first time. (That is, it might be delayed before it has a chance to complete its increment operation.) Since there are only k registers, and each increment

operation must change the value of at least one register, the value of at least one register, say r, is changed at least two times. Let p_1 be the process that was the first to change r and let p_2 be the process that was the second to change r.

Since $k+1$ processes participated in α, there are $n_0 - (k+1) = 2n_1 - 1$ processes that have not yet participated in α. Let G_1 be the set of processes that have not yet participated in α together with the process p_2. We divide G_1 into two groups H_1^1 and H_2^1 where $|H_1^1| = |H_2^1| = n_1$ and H_2^1 includes p_2.

Next we construct the run ρ_1 as follows. First we activate the processes exactly as in α until the point where p_1 is about to change r for the first time. Then we suspend p_1, and activate each of the processes in H_1^1 until it is also about to change r. This will happen since process p_1 is identical to each of the processes in H_1^1. We then suspend the processes in H_1^1, let p_1 change the value of r, and let the run continue as in α, until the point where p_2 is about to change r for the second time. Next we activate each of the other processes in H_2^1 until it is also about to change r. This will happen since process p_2 is identical to each of the processes in H_2. Notice that so far no process in G_1 has written in the shared memory.

We now activate the processes in H_2^1 and H_1^1 in alternation, until each process flips r one time. That is, first we pick a process from H_2^1 and let it flip r and then pick a process from H_1^1 and let it flip r back, and so on until each process in these two groups flips r one time.

Notice that between the point where a process in H_1^1 is suspended and the point when it is activated the value of r is changed an even number of times and hence the process will not notice that r has been changed and will change r when it is activated again, and similarly for processes in H_2^1. Next we let the k processes not in G_1, that might be in the middle of an increment operation, complete their increment operation. Finally, if each process in H_1^1 can complete its increment operation without flipping any more of the counter bits then we let all these processes do it, otherwise (i.e., *no* process in H_1^1 can complete its increment operation without flipping another bit), they take no more steps; this procedure is repeated for the processes in H_1^2. The resulting run is ρ_1.

Let ρ_1' be the the subsequence of all events in ρ_1 involving processes not in G_1. In ρ_1, each change of r by a process in H_2^1 is immediately followed by a change of r by a process in H_1^1, and hence all operations by processes in G_1 are invisible by processes not in G_1. That is, the runs ρ_1 and ρ_1' are *indistinguishable* for processes not in G_1 and hence ρ_1' is a complete legal run, and the value of the counter in ρ_1 and ρ_1' is the same. All this implies that the group of processes G_1 is *hidden* in run ρ_1.

Round $i+1$: Assume that we can construct run ρ_i which has the following properties: there is a group G_i of processes that is *hidden* in ρ_i, each process in G_i has already flipped the counter registers i times or have completed one increment operation in ρ_i, all processes not in G_i are idle, and $G_i = \bigcup_{j \in \{1,...,2^i\}} H_j^i$ where for each $j \in \{1,...,2^i\}$ all the processes in H_j^i have exactly the same history in ρ_i and $|H_j^i| = n_i$. (When i is 0 or 1, we have already shown how to construct ρ_0 and ρ_1 with these properties.)

We next show how to build a run ρ_{i+1} with such properties.

Let α_1^i be a run similar to α, in which all the processes in H_1^i is activated in a row, each of them is activated until it either changes the value of a counter register, or it terminates its increment operation. (Note that once a process $p \in H_1^i$ terminates its increment operation without changing a bit, all the remaining processes in H_1^i, being identical to p, will do the same). We consider two cases:

(a) In α_1^i at most k processes change the value of a register before terminating their increment operation (this includes the case where all the processes in H_1^i already terminated their increment operations in previous stages of the construction). In this case, we let activate the first k processes in H_1^i until each of them either terminates or changes the value of one bit, and then let all the remaining processes terminate their increment operations (without

changing the value of any register). Then we let the first k processes from H_1^i complete their increment operation, and split the rest of the processes into two disjoint groups. These two groups are H_1^{i+1} and H_2^{i+1} where $|H_1^{i+1}| = |H_2^{i+1}| = \frac{n_i - k}{2} = n_{i+1}$.
(b) At least $k+1$ processes change the value of a register in α_1^i. In this case, starting from ρ_i, we first activate only the processes in H_1^i as described in Round 1, where the groups G, H_1^1 and H_2^1 in Round 1 corresponds to H_1^i, H_1^{i+1} , H_2^{i+1}, respectively, and ρ_o in Round 1 corresponds to ρ_i. That is, we extend ρ_i to a run ρ_{i_1} by activating only processes from H_1^i. In doing so, $|H_1^i| - k$ processes in H_1^i are divided into two groups H_1^{i+1} and H_2^{i+1} of the same size, such that all the processes in each of these groups has exactly the same history in ρ_{i_1}. Again, the the size of each of this two groups is $|H_1^{i+1}| = |H_2^{i+1}| = \frac{n_i - k}{2} = n_{i+1}$.
The other k processes from H_1^i complete their increment operation and are idle at ρ_{i_1}.

We repeat doing so sequentially with H_2^i, H_3^i and so on. After repeating this procedure for 2^i times we get the run ρ_{i+1} as required. That is: We have the group $G_{i+1} = \bigcup_{j \in \{1,...,2^{i+1}\}} H_j^{i+1}$ where for each $j \in \{1, ..., 2^{i+1}\}$ all the processes in H_j^{i+1} have exactly the same history in ρ_{i+1} and $|H_j^{i+1}| = n_{i+1}$. In addition, the group G_{i+1} is *hidden* in ρ_{i+1}, each processes in G_{i+1} has already flipped the counter registers $i+1$ times, and all processes not in G_{i+1} are idle in ρ_{i+1}.

Round f: In Round f we get the run ρ_f. Here $G_f = \bigcup_{j \in \{1,...,2^f\}} H_j^f$ where for each $j \in \{1, ..., 2^f\}$, $|H_j^f| = n_f = 1$. Since $f = flip(Pr)$, all the processes in G_f have completed their increment operation and hence ρ_f is a *complete* run. Thus, we have constructed a complete run in which a group of 2^f processes is hidden, where each process in this group is idle, and it has completed one increment operation. □

4 Main result

In this section we use the Hiding Lemma to show that in a model where the basic atomic operations are performed on binary registers (bits), for any wait-free counting protocol Pr (modulo m), if $flip(Pr)$ is finite[3] then m must divide $2^{flip(Pr)}$. This implies that in such a model it is possible to count only modulo powers of 2.

Lemma 4.1 *Let Pr be a wait-free n-counting protocol, let $k = space(Pr)$ and let $f = flip(Pr)$. If $k \leq \frac{n-2^f}{2^f-1}$ then m divides 2^f. Thus, if $k \leq \frac{n-2^f}{2^f-1}$ then $f \geq log(m)$ and m is a power of 2 (which implies that if m does not divides 2^f then $k > \frac{n-2^f}{2^f-1}$).*

Proof: If $k \leq \frac{n-2^f}{2^f-1}$ then by the Hiding Lemma there exists a complete run ρ and a group of 2^f processes G_f, such that G_f is hidden in ρ, and each process in G_f has completed one increment operation.

Let ρ' be the subsequence of all events in ρ involving processes not in G_f. The fact that G_f is *hidden* in ρ, implies that ρ' is a complete run and that the values of the counter in ρ and ρ' is the same (modulo m).

Since the counter is incremented in ρ exactly 2^f times more than in ρ', the fact that the value of the counter in ρ and ρ' is the same (modulo m) implies that m must divide 2^f. □

In the case where the number of processes is not bounded the following theorem, which is the main result of this section, follows from Lemma 4.1.

Theorem 4.1 *Let Pr be a wait free counting protocol which counts modulo m, and assume that both $space(Pr)$ and $flip(Pr)$ are finite. Then m divides $2^{flip(Pr)}$. Thus, $flip(Pr) \geq log(m)$ and m is a power of 2.*

[3]Note that even if $flip(Pr)$ is finite, Pr is not necessarily wait free, and vice versa.

Proof: Let $k = space(Pr)$ and $f = flip(Pr)$. Since both f and k are finite, there is an integer n' such that $k \leq \frac{n'-2^f}{2^f-1}$. Since Pr is an n-counting protocol for every n, it is in particular an n'-counting protocol. Thus, Pr satisfies the assumptions of Lemma 4.1 (for $n = n'$). The theorem follows. □

In the theorem, we get that $flip(Pr) \geq \log(m)$. An immediate question is whether there exists a counting protocol where $flip(Pr) = \log(m)$. The answer is positive, since in the Positional counter presented in [MTY92] $flip(Pr) = \log(m)$ and it works for any number of concurrent increment operations.

Next we show how interesting results about counting networks which are a special type of counting protocols, can be easily derived from Theorem 4.1.

5 Consequences for Counting and Smoothing Networks

Counting networks, introduced in [AHS91], are a new class of networks that can be used to count. They are constructed from simple two-input two-output computing elements called 2-balancers (abbv. balancers), connected to one another by wires. A balancer repeatedly sends the inputs it receives, one to the top and one to the bottom, and can be implemented by a read-modify-write bit. An increment operation is performed by placing a token on an input wire. The token traverses a sequence of balancers, and leaves on an output wire. The output (input) wires are numbered from 0 to $m-1$. Counting networks are required to satisfy the following *step* property: denote by o_i the number of tokens that traversed the i-th output wire. Then at quiescent states, for each $0 \leq i < j \leq m-1$, $0 \leq o_i - o_j \leq 1$.

A counting network can be used to implement a counting protocol as follows: use one input wire to insert tokens, and let the value of the counter be the index of the output wire through which a token that is inserted to the counting network will leave it. Thus, the result proved in Theorem 4.1 applies to counting networks, which means that a counting network over m output wires is possible only when m is a power of 2, and its depth is at least $\log m$.[4]

A smoothing network is a balancing network which is a generalized form of a counting network. It may be used as a building block in constructing a counting network. It satisfies the following weaker property: at quiescent states, for each $0 \leq i, j \leq m-1$, $|o_j - o_i| \leq 1$. Clearly, every counting network is a smoothing network, but not vice versa. More generally, for any integer $K \geq 1$, a *K-smoothing network* is a balancing network which satisfies the following property: at quiescent states, for each $0 \leq i, j \leq m-1$, $|o_i - o_j| \leq K$. Unlike counting network, a K-smoothing network is not necessarily isomorphic to a sorting network even for $K = 1$ [AHS91].

Using the combinatorial properties of smoothing networks, we provide below a simple and direct proof that for every K, a K-smoothing network (and hence also a counting network) requires $\log m$ depth: Let $\mathcal{S}$ be a K-smoothing network (mod m). Consider a scenario in which Km tokens are inserted via the *same* input wire. By the definition of a K- smoothing network, these tokens must leave on m distinct output wires. Thus, the tokens travel m distinct paths. Since the output degree of each balancer is 2, at least one of these paths must have at least $\log_2 m$ edges.

Next, we show that our result implies that a similar lower bound applies to a considerably larger class of protocols, in which there is no restriction on the way the tokens traverse the network, neither on the size of the local memory of the processes, nor on the number of read operations that processes may perform for processing a token. As before, we assume a shared memory model which supports *rmw* operations on shared bits.
A *K-smoothing protocol* is a protocol in which tokens are inserted, by many processes, in a

[4]The original proof of the $\log m$ lower bound for counting networks is based on the observation that every counting network is isomorphic to a sorting network [AHS91].

given "input" location, and are then distributed among m "output" locations. The protocol is required to satisfy the following property: Let o_i denote the number of tokens distributed to the $i-th$ output location; then in quiescent states (ie, states in which all the tokens that were inserted are already distributed), $|o_i - o_j| \leq K$.

COROLLARY 5.1 *Let $\mathcal{S}$ be a wait-free, K-smoothing protocol with m output locations. If both space($\mathcal{S}$) and flip($\mathcal{S}$) and finite, then m is a power of 2 and flip($\mathcal{S}$) $\geq \log m - \lceil \log (2K+1) \rceil$.*

Proof: A K-smoothing protocol can be transformed into a counting protocol by adding $\lceil \log(2K+1) \rceil$ bits at each output location. These bits enable the simulation of a counter (mod M) on each output location, where M is the least power of 2 which is larger than $2K$.[5] By using a counter (mod M) it is possible to distinguish at quiescent states those output locations that carried the smallest number of tokens (not the number itself), and for each location, how many tokens it has carried in addition to this smallest number. We can then sum up the number of additional tokens (over all locations), and the value of the counter is the number of these additional tokens modulo m, where m is the total number of output locations. The result follows by Theorem 4.1. □

In [AA92] it is proven that the number of output wires of any acyclic K-smoothing network is a power of 2. It is shown how to implement counting networks where the number of output wires is not a power of two, by using cyclic counting networks. These implementations has the unpleasant property that a token my traverse a cycle in the network forever, never finding its way out. It follows from Corollary 5.1 that in the implementation of any K-smoothing protocol, the processing of a token cannot terminate within a fixed number of bit changes, unless the number of output locations is a power of 2.

REFERENCES

[AA92] E. Aharonson and H. Attiya. Counting networks with arbitrary fan-out. In *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 104–113, January 1992.

[AHS91] J. Aspnes, M. Herlihy, and N. Shavit. Counting networks and multi-processor coordination. In *Proc. 23rd ACM Symp. on Theory of Computing*, pages 348–358, May 1991.

[HBS92] M. Herlihy, Lim. B., and N. Shavit. Low contention load balancing on large-scale multiprocessors. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, July 1992.

[HSW91] M. Herlihy, N. Shavit, and O. Waarts. Low contention linearizable counting. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pages 526–535, October 1991.

[KP92] M. Klugerman and C. Plaxton. Small-depth counting networks. In *Proc. 24rd ACM Symp. on Theory of Computing*, pages 417–428, October 1992.

[MTY92] S. Moran, G. Taubenfeld, and I. Yadin. The distributed counter problem. In *Proc. 11th ACM Symp. on Principles of Distributed Computing*, pages 59–70, August 1992.

[5] This counter can be implemented by using the Positional counter of [MTY92], which allow to read-modify-write one bit at a time and guarantees that the counter shows the correct value at quiescent states.