A Lower Bound on Wait-Free Counting*

Shlomo Moran† Gadi Taubenfeld‡

June 27, 1996

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, $\log m = f$ for some integer $f \leq flip(Pr)$. Thus, $flip(Pr) \geq \log m$ and m is a power of 2. By a result of [MTY92] the above lower bound on $flip(Pr)$ is tight.

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

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 processes that communicate 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.

*A preliminary version of this work is to appear in the *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing*, Ithaca, August 1993.

†CWI, P.O. Box 4079, 1009 AB Amsterdam, the Netherlands and Computer Science Department, Technion, Haifa 32000, Israel. Part of this work was done while the author was visiting at AT&T Bell Laboratories.

‡The Open University, 16 Klausner st., P.O.B. 39328, Tel-Aviv 61392, Israel. Part of this work was done while the author was working at AT&T Bell Laboratories.

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 returned by the look operation is in the cyclic interval $[\ell, r]$ mod m, which is the set $\{i(\text{mod} m) | i \in \{\ell, \ell+1, \cdots, r\}$.

We use the notion *counting protocol* for an implementation of a counter which satisfies the weak correctness requirement mentioned earlier, and which counts 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 $space(Pr)$ be the number of shared registers used for the counter. Notice that both $flip(Pr)$ and $space(Pr)$ might be infinite. 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 $space(Pr)$ is finite. Then $\log m = f$ for some integer $f \leq flip(Pr)$. 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 $depth(Pr) = flip(Pr) \geq \log m$.

[1] All logarithms are to base two.

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 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, in the worst case, at least $\log m - \lceil \log(2K+1) \rceil$ bit changes for distributing a single token. 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 requires $\Theta(m \log m)$ bits.

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

Finally, we point out that the result stated in Proposition 3.1 is in some respects 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, HLS92, HSW91, KP92]. A tight bound of $\Theta(m \log m)$, on the number of 2-balancers needed to construct counting networks is proved 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 modulo 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]. A concurrent counter (mod m) holds an integer from $0, .., m-1$, and 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 guarantee only that at quiescent states, a `look` operation returns the correct value, 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. That is, for a given look operation, let ℓ be the initial value of the counter plus the number of increment operations that were completed before the look operation started, and let r be the initial value of the counter plus the total number of increment operations that were initiated before the look operation was completed. Then the look operation should return some value between ℓ and r (mod m).

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

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 $(\textsf{increment}, val, \vec{v}_{init})$, 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 $val(\vec{v}_{init}) = 0$.

[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.

A process performs the increment operation on the counter by executing an **increment** procedure. (Many increments can take place concurrently.) Unless otherwise stated, we assume that all the processes are identical. In case the processes have unique identifiers, the identifier of a process can be passed as an argument to the increment procedure. Hence, in such a case, different processes may increment the counter in a different way. Each increment operation consists of a sequence of atomic read-modify-write (in short *rmw*) steps applied to the counter registers, where the counter registers consist of *all* the shared memory registers that are accessed during all possible increment operations. 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* of a counting protocol starts from some initial values of the counter registers and consists of a finite or infinite sequence of atomic *rmw* steps, which are executed by some of the processes; each process activated in a run is repeatedly executing a sequence of (one or more) increment operations, the last of which may be incomplete. We denote by *initial(x)* the contents of the counter registers at the beginning of the run x, and by *final(x)* the contents of the counter at the end of x. A run x is *legal* if $initial(x) = \vec{v}_{init}$. A process 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 *idle* in a given run if it has completed all its increment operations in that run. 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 involved in an increment operation in it.

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

correctness 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 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

Informally, in this section we show that if no more than f flips are performed during an increment operation, then there exists special runs in which processes are "hidden". More precisely, let $\vec{v}$ be a vector, and let $first(\vec{v})$ denote the supremum on the number of times a process may flip a counter bit during its first increment operation in a run x of Pr where $inital(x) = \vec{v}$. (In particular, $first(\vec{v}_{init}) \leq flip(Pr)$). We show that if $f = first(v)$ is finite, then 2^f increment operations, each of which executed by a different process, can be hidden in some run x with $init(x) = \vec{v}$. 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^f increment operations actually took place.

In the next section we characterize a set of vectors such that for every vector $\vec{v}$ in this set $first(\vec{v})$ is finite, and in the following section we combine these two results to prove our main result, which does not assume any bound on $flip(Pr)$.

In the next two sections we will assume that all the processes are identical. In Subsection 5.1, we will show how this assumption can be removed by using more processes, and assuming that $flip(Pr)$ is finite.

DEFINITION **1** *A group of processes G is* hidden *in a 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.*

Recall, that the counter consists of *all* the shared memory registers that are accessed during all possible increment operations, and that for a counting protocol Pr, $space(Pr)$ denotes the size of the counter (number of shared bits) used by Pr.

Lemma 3.1 *(The Hiding Lemma) Let Pr be a wait-free counting protocol, let $k = space(Pr)$, let $\vec{v}$ be a vector such that $f = first(\vec{v})$ is finite, and let $n = k(2^f - 1) + 2^f$. Then there exists a complete run, ρ, such that $initial(\rho) = \vec{v}$, at most n processes are activated in ρ, and a group of 2^f processes, each of which performs one increment operation, is hidden in ρ.*

Proof: 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 $n_0 = n$, $n_f = 1$ 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: $initial(\rho_i) = \vec{v}$, 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 either had already completed its increment operation and is idle, or it has already flipped the counter registers i times. The run ρ_f is ρ. Notice that since $initial(\rho_f) = \vec{v}$, all the 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, $initial(\rho_0) = \vec{v}$, and G_0 includes n_0 processes. 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. (Thus, at this point p_1 have not yet changed any shared register.) 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. (Recall again that the counter consists of all the shared memory registers that are accessed during all possible increment operations.)

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 $G_i = \bigcup_{j \in \{1,\ldots,2^i\}} H_j^i$ 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 these 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 $G_{i+1} = \bigcup_{j\in\{1,\ldots,2^{i+1}\}} H_j^{i+1}$ 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 $G_f = \bigcup_{j\in\{1,\ldots,2^f\}} H_j^f$ where for each $j \in \{1, ..., 2^f\}$, $|H_j^f| = n_f = 1$.

Since no process performs more than f flips in it first increment operation, 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. ∎

We now state a result that follows easily from Lemma 3.1. 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 the following result 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 at most 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).

Proposition 3.1 *Let Pr be a wait-free n-counting protocol, let $k = space(Pr)$ and assume that $f = first(\vec{v}_{init})$ is finite. If $n \geq k(2^f-1)+2^f$ then m divides 2^f. Thus, if $n \geq k(2^f-1)+2^f$ then $f \geq \log m$ and m is a power of 2.*

Proof: If $n \geq k(2^f-1)+2^f$ then by the Hiding Lemma (with $\vec{v} = \vec{v}_{init}$), there exists a complete legal run ρ in which at most n processes are involved, and a group G_f of 2^f

processes is hidden in ρ, where each process in G_f has completed one increment operation in ρ.

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' are 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. ∎

Both lemma 3.1 and Proposition 3.1 are proved under the assumption that $first(\vec{v}_{init})$ is finite. We show elsewhere that there are wait-free counting protocols in which $first(\vec{v}_{init}) = \infty$, which raises the question whether the conclusions of Proposition 3.1 are valid also without assuming that $first(\vec{v}_{init})$ is finite. In the next two sections we show that this is indeed the case.

4 Bounding the Number of Flips

The main result of this section is that for some vectors $\vec{v}$ which is *reachable* (see definition below) from $\vec{v}_{init}$, $first(\vec{v})$ is finite.

A vector $\vec{v}$ is *reachable* from a vector $\vec{u}$ (w.r.t. a counting protocol Pr) if there is a run x (of Pr) with $initial(x) = \vec{u}$ and $final(x) = \vec{v}$. (Recall that *initial*(x) denotes the contents of the counter registers at the beginning of a run x, and *final*(x) denotes the content of the counter registers at the end of a run x.) Notice that it is not required that x is a complete run (i.e., a run where all processes are idle). The *run graph* of Pr is a directed graph $G = (V, E)$, where V is the set of all k-ary binary vectors, and $E = \{(\vec{u}, \vec{v}) : \vec{v}$ is reachable from $\vec{u}\}$.

A strongly connected component C of G is *maximal* if there is no edge from a vertex in C to a vertex not in C. A vertex in G is maximal if it belongs to a maximal strongly connected component. Let $G = (V, E)$ be the run graph of Pr. If V is finite, G must contain a maximal strongly connected component, and since G is transitive, from every vertex in G there is an edge to a maximal vertex in G. In particular, there is a maximal vector $\vec{v}$ which is reachable from $\vec{v}_{init}$. (Recall that $first(\vec{v})$ denotes the supremum on the number of times a process may flip a counter bit during its first increment operation in a run x of Pr where $inital(x) = \vec{v}$.)

Lemma 4.1 *Let Pr be a wait-free counting protocol, where space(Pr) is finite and all the processes are identical, let G be the run graph of Pr, and let $\vec{v}$ be a maximal vector in G which is reachable from $\vec{v}_{init}$. Then $first(\vec{v})$ is finite.*

Proof: Let C be the maximal strongly connected component of G that contains $\vec{v}$. We start by defining T_C, which is the binary tree that describes the program for the first increment operation of a process in a run which starts when the contents of the counter is any vector in C.

The root of T_C is the initial local state of the process, and each vertex v in it corresponds to a local history of the process executing its first increment . (Recall that all processes are identical.) We point out that the local history of a process determines what is the next shared register this process is going to access (if any). Each vertex v of T_C is either a leaf, meaning the increment is already completed, or it is associated with the next register, $reg(v)$, on which the process is going to perform a *rmw* operation.

We say that a register r is *redundant* in C if the value of r is fixed in C. If $reg(v)$ is not redundant in C, then the vertex v has two descendants: one for the case where the value of $reg(v)$ is 0, and the other for the case it is 1. Otherwise, v has exactly one child, corresponding to the fixed value of $reg(v)$ in C.

Note that $first(\vec{v})$ is bounded by the depth of T_C. Thus, it suffices to show that T_C is a finite tree. To show that, we assume that T_C is infinite, and reach a contradiction. Assuming that T_C is infinite, by the infinity lemma, T_C contains an infinite path, π. We next construct a legal run z in which a process p makes its first increment operation along the infinite path π, and thus never completes its first increment – contradicting the wait-freedom property.

The legal run z is constructed as follows. It starts with a run z_0 where $initial(z_0) = \vec{v}_{init}$ and $final(z_0) = \vec{v}$ (such a run exists since $\vec{v}$ is reachable from $\vec{v}_{init}$).

Let p be any process which is not activated in z_0. We now extend z_0 (and construct z) in phases. In phase i we have the finite run z_i in which process p already made i steps , corresponding to the first i edges in π. Let $\vec{v}_i = final(z_i)$, and assume that the $i+1$-th edge of π corresponds to reading a value b in a register r. By the maximality of C, $\vec{v}_i$ belongs to C. Thus it is possible to extend z_i without activating p to a run y_i, such that the contents of r in y_i is b. The run z_{i+1} is obtained by letting p take one step at the end of y_i.

In z process p takes infinite number of steps and never completes it first increment operation, contradicting the wait-freedom property. Thus, the binary tree T_C must be finite. ∎

Notice that if we omit the assumption that only the first increment operation is considered, then the above lemma becomes incorrect. It is easy to device a counting protocol (in which each process has unbounded private memory), in which for each run x and for each i, the i'th increment operation of a process in x takes at least i flips.

5 Main result

In this section we use the Hiding Lemma and Lemma 4.1 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), there exists an integer $f \leq flip(Pr)$ such that m divides 2^f. This implies that in such a model it is possible to count only modulo powers of 2, and that $flip(Pr) \geq \log m$. In this section it is assumed that the number of processes is not bounded.

Theorem 5.1 *Let Pr be a wait-free counting protocol which counts modulo m, assume that $space(Pr)$ is finite, and that all the processes are identical. Then, $\log m = f$ for some integer $f \leq flip(Pr)$. Thus, $flip(Pr) \geq \log m$ and m is a power of 2.*

Proof: Let $f = \min\{first(\vec{u}) : \vec{u} \text{ is a maximal vector reachable from } \vec{v}_{init}\}$. f is finite by Lemma 4.1. Let $\vec{v}$ be a maximal vector reachable from $\vec{v}_{init}$ with $first(\vec{v}) = f$. The theorem is proved by constructing a complete legal run ρ in which 2^f processes are hidden as follows.

ρ starts with a run x where $initial(x) = \vec{v}_{init}$ and $final(x) = \vec{v}$ (x exists since $\vec{v}$ is reachable from $\vec{v}_{init}$). Let S be the set of processes that have not terminated yet in x.

Since in any run starting from $\vec{v}$ a process performs at most f flips during its first increment operation, the Hiding Lemma implies that there exists an extension y of x, and a group of 2^f processes G_f ($G_f \cap S = \emptyset$), such that G_f is hidden in y, and each process in G_f has completed one increment operation. Finally, we extend y to a complete run ρ by letting all the processes in S complete their increment operations in some way.

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 legal 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. ∎

Notice that while f in the statement of the theorem is finite $flip(Pr)$ might be infinite. From 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.

5.1 Unique Identifiers

So far we have assumed that the processes are identical. Next we explain how our main result can be proved without this assumptions, provided we are given that $flip(Pr)$ is finite. That is, Theorem 5.1 holds even if all the processes have unique identifiers, provided $flip(Pr)$ is finite (actually, we only need that $first(\vec{v}_{init})$ is finite).

Theorem 5.2 *Let Pr be a wait-free counting protocol which counts modulo m, assume that both space(Pr) and $flip(Pr)$ are finite, and that the processes have unique identifiers. Then,* $\log m = f$ *for some integer* $f \leq flip(Pr)$. *Thus,* $flip(Pr) \geq \log m$ *and m is a power of 2.*

Proof: The assumption that the processes are identical is used only in the proof of the Hiding Lemma. We show how the proof of the Hiding Lemma can be modified to deal also with the case where processes have unique identifiers. In the construction in the Hiding Lemma it is shown that, given a set of processes G which is hidden in a run x and where all the processes in G have the same history and are involved in increment operations, we can extend x to a run y such that: two subsets G_1 and G_2 of G are hidden in y, all the processes in G_i ($i = 1, 2$) have the same history and have flipped one more bit in $(y - x)$, and $|G_i| \geq \frac{|G|-k}{2}$ (where $k = space(Pr)$). We achieve it by activating $\leq k+1$ processes of G, until each one of them flips one bit (or terminates the increment operation). If no process terminates its increment operation, then one bit has been flipped twice. We use this fact about the bit and the assumption that the processes are identical to create the two subsets G_1 and G_2.

The above construction of y can be achieved without assuming that the processes are identical, in the price of a weaker lower bound on the size of G_1 and G_2 - namely, $G_i \geq \frac{|G|-k-1}{2k+2}$, $i = 1, 2$. At any point in time (that is for any given global state) we can partition G into $k+1$ sets such that each of the processes in the same set, when activated alone, will change the value of the same shared register (counter bit) first, or will terminate its increment operation without flipping any bit. We notice that the size of at least one such set is at least $\frac{|G|-k-1}{k+1}$.

Now, we construct y without assuming that the processes are identical. This is done by properly choosing the $k+1$ processes that are activated in order to find a bit that is flipped twice. Whenever we need to activate one more process (of the $k+1$ processes) we look at the partition of G with respect to the current global state (as explained above) and activate a process from the biggest partition set. This guarantees that for the two processes p_1 and p_2 that have flipped the same bit, there are at least $\frac{|G|-k}{k+1}$ processes in G that will behave as p_1, when activated from the same state, and there are at least $\frac{|G|-k-1}{k+1}$ processes that will behave as p_2 (the two sets are not necessarily disjoint). Hence there are two disjoint sets G_1 and G_2, each has at least $\frac{|G|-k-1}{2k+2}$ processes, which behave as p_1 and p_2 respectively. Thus, we can now continue as in the construction for the case when the processes are identical, starting with a larger set of processes. ∎

Next we show how interesting results about counting networks which are a special type of counting protocols, can be easily derived from Theorem 5.1.

6 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 at a given state be the index of the output wire through which a token that is inserted to the counting network in this state will leave it. Thus, the result proved in Theorem 5.1 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 $\log m$.[3]

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].

[3]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].

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 $\log m$ 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 rmw 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 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 (i.e., states in which all the tokens that were inserted are already distributed), $|o_i - o_j| \leq K$.

Corollary 6.1 *Let $\mathcal{S}$ be a wait-free, K-smoothing protocol with m output locations, and assume that space$(\mathcal{S})$ is finite. Then,* $\log m = f$ *for some integer* $f \leq flip(\mathcal{S})+\lceil\log\ (2K+1)\rceil$.

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 f) on each output location, where f is the least power of 2 which is larger than $2K$.[4] By using a counter (mod f) 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 5.1. ∎

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 6.1 that there is no wait-free implementation of any K-smoothing protocol, unless the number of output locations is a power of 2. Thus Corollary 6.1 is a strict generalization of a result in [AA92].

7 Wait-Free Counting with Bounded Memory

In this section we show that if each of the processes is assumed to be a finite state machine (that is, the local memory of each process is bounded) then the following version of Theorem 5.1 is valid:

[4]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.

Theorem 7.1 *Let Pr be a wait free counting protocol which counts modulo m. If each process is a finite state machine with at most B states, then m divides 2^B. Thus, $B \geq \log m$ and m is a power of 2.*

Proof: The proof is based on modifications of the construction in the proof of Lemma 3.1. We show that there exists a complete run, ρ, in which a group of 2^B processes is hidden, and each process in this group has completed one increment operation and is idle in ρ. The result then follows. First, observe that since each process has at most B states it can access at most B shared registers (the state of a process determines uniquely which shared register it is going to access in its next atomic step). Thus, assuming that the processes are identical, $space(Pr)$ is bounded by B. In the sequel, we use k do denote $space(Pr)$.

Let M be a natural number and $\vec{v}$ be a binary k-vector. Then $\rho(\vec{v}, M)$ is a run constructed as the run ρ in Lemma 3.1, where the initial vector is assumed to be $\vec{v}$ and the number of rounds is M. (The run ρ of Lemma 3.1 is $\rho(\vec{v}, first(\vec{v}))$. Note that if $M < first(\vec{v})$, then it may be that not all the processes in $\rho(\vec{v}, M)$ are idle; however, by the construction 2^M processes, each of which has started one increment operation, are hidden in the run $\rho(\vec{v}, M)$.

Let $Pr = (\textsf{increment}, val, \vec{v}_{init})$ be a counting protocol where each process has at most B internal states. Recall that in the run graph of Pr (defined in Section 4), there exists a maximal vector $\vec{v}$ which is reachable from $\vec{v}_{init}$. That is, there is a run x where $initial(x) = \vec{v}_{init}$, $final(x) = \vec{v}$ and $\vec{v}$ is maximal for Pr.

We prove that no process which is activated in $\rho(\vec{v}, B+1)$ executes more than B flips. Assume to the contrary that there exists some process p which has performed $B+1$ flips in $\rho(\vec{v}, B+1)$. We show that this yields a contradiction, by constructing a run in which the process p is activated infinitely often but never completes a single increment operation.

Since p has at most B states and it performs $B+1$ flips during $\rho(\vec{v}, B+1)$, there are some $1 \leq i < j \leq B+1$, such that the i'th flip and the j'th flip of p in $\rho(\vec{v}, B+1)$ are done when p is in the same state. A run τ in which p continuously repeats the steps between its i'th flip to its j'th flip is constructed as follows:

$$\tau = x \cdot y \cdot z \cdot (\sigma_1 \cdot z_1) \cdot (\sigma_2 \cdot z_2) \cdots$$

where:

- x is the run mentioned above, for which $initial(x) = \vec{v}_{init}$ and $final(x) = \vec{v}$ for some maximal vector $\vec{v}$.
- The run y is a prefix of $\rho(\vec{v}, B+1)$. It follows $\rho(\vec{v}, B+1)$ up to the point where p is about to perform its j'th flip (i.e., the last step in y is the one which precedes the j'th flip by p in $\rho(\vec{v}, B+1)$).
- Let $final(y) = \vec{u}$. Then for $\ell = 1, 2, \cdots$ the run z_ℓ is a run where $initial(z_\ell) = \vec{u}$ and $final(z_\ell) = \vec{v}$, and the processes activated in each z_ℓ are distinct. Note that z_ℓ exist since $\vec{v}$ is a maximal vector.
- For $\ell = 1, 2, \cdots$ the run σ_ℓ is similar to $\rho(\vec{v}, B+1)$, except that process p is activated in each σ_ℓ between its i'th and j'th flips; all other processes activated during y and each σ_ℓ are distinct.

Thus, τ is a run in which process p is activated infinitely often but never terminates its increment operation, a contradiction.

We have shown that no process activated during $\rho(\vec{v}, B+1)$ executes more than B flips. Thus, all the processes in the group G_B, constructed at Round B during the construction of $\rho(\vec{v}, B+1)$, have already completed their increment operation and are idle. Hence, we get that in the run $\rho(\vec{v}, B)$ there are 2^B hidden processes, each of which has completed one increment operation, and further more $\rho(\vec{v}, B)$ is complete (i.e., all the processes are idle).

Consider the run $\rho = x \cdot \rho(\vec{v}, B) \cdot x'$, where x is, as before, a run where $initial(x) = \vec{v}_{init}$ and $final(x) = \vec{v}$; and x' is a run in which we let all the active processes in $x \cdot \rho(\vec{v}, B)$ complete their increment operation (in fact x' includes only steps of processes that were activated in x). Then ρ is a complete legal run in which 2^B increment operations are "hidden". The proof is completed in a way similar to the proof of Theorem 5.1. ∎

Acknowledgement We thank Hagit Brit for helpful discussions which helped us in improving our results.

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.

[HLS92] M. Herlihy, B. Lim, 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. Concurrent counting. In *Proc. 11th ACM Symp. on Principles of Distributed Computing*, pages 59–70, August 1992. To appear in *Journal of Computer and System Sciences.*